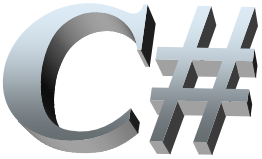


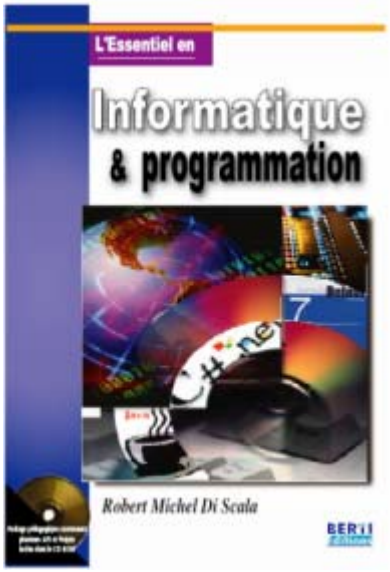
Programmer dans .Net Framework avec le langage C# version 2.0

713 pages de cours + 94 pages d'exercices corrigés

Une grande partie du contenu de ce livre pdf de cours d'initiation à C#, est inclus dans un ouvrage papier de 1372 pages édité en Novembre 2004 par les éditions Berti à Alger.

http://www.berti-editions.com

L'ouvrage est accompagné d'un CD-ROM contenant les assistants du package pédagogique.



Rm di Scala

Mise à jour d'Août 2006

Pour les mises à jour et les autres cours pdf du même auteur, consultez le site suivant : http://www.discala.net

SOMMAIRE

Types, opérateurs, instructions

Introduction ……..…. ……. **P.3**
Les outils élémentaires ……..…… **P.4**
Les éléments de base ……...……. **P.12**
Les opérateurs + exemples ……. **P.21**
Les instructions …….……. **P.35**
Les conditions ……. **P.40**
Les itérations ……. ……. **P.46**
Les ruptures de séquence ……. ……. **P.50**
Classes avec méthodes static …….……. …… **P.54**

Structures de données de base

Classe String ……..…. **P.72**
Tableaux, matrices ……..…. **P.80**
Collections, piles, files, listes …….……..…. **P.95**

C# est orienté objet

Classes, objets et méthodes ……..…. **P.117**
Polymorphisme d'objet …….……..…. **P.151**
Polymorphisme de méthode ……..….……..…. **P.163**
Polymorphisme d'interfaces ……..…. **P.188**
Classe de délégation …….……..…. **P.212**
Traitement des exceptions …….……..…. **P.228**
Processus et multi-threading …….……..…. **P.251**

IHM avec C#

Programmation événementielle ……..……. **P.279**
Les événements en C # ……. **P.303**
Propriétés et indexeurs …….……...… **P.322**
Fenêtres et ressources mémoires ……..…….…… **P.348**
Contrôles dans les formulaires ……..……... **P.386**
Exceptions comparées à Delphi et java ……..…….. **P.410**
Données simples flux et fichiers ……..……. **P.414**

Eléments principaux depuis la version C# 2.0

Les generics ……..….…… **P.447**
Les classes partielles … ….…….….. **P.457**
Les méthodes anonymes ……..……..…….. **P.465**
TAD de liste, pile, file et classes génériques …….……..…… **P.476**
TAD d'arbre binaire et classes génériques …….……..….… **P.494**

Principes des bases de données relationnelles …….. P.535
ADO.Net et données relationnelles ……..…….. P.561
ADO.Net et SQL serveur 2005 ……..…….. P.574

Programmation web avec ASP.net 2.0

Page web écrite uniquement avec des contrôles HTML simples ……P.617
Page web écrite uniquement avec des contrôles serveurs…… ……P.630
Communication entre pages web…… ……...P.649
Page web et BD avec GridView……. …….……P.660
Cookie dans une page web ……. …….……...…P.678

Bibliographie …….…….…..
Exercices …….……..
94 pages d'exercices avec solutions

Pour pouvoir s'initier à C# avec ce cours et à peu de frais dans un premier temps, il faut télécharger gratuitement sur le site de Borland, l'environnement Borland studio Delphi 2006 édition personnelle, ou aller sur le site de Microsoft et télécharger gratuitement Visual C# express, ou bien utiliser la denière version de l'environnemnt open source Sharpdevelop qui fonctionne sous Net framework run-time..

Remerciements : (*pour les corrections d'erreurs*)
A tous ceux qui m'ont envoyé un petit mail me signalant une erreur, une omission …

A mon épouse Dominique pour son soutien et sa patience qui me permettent de consacrer de nombreuses heures à la construction du package et des cours inclus et surtout qui a eu la constance de relire entièrement toutes les pages de la version initiale de l'ouvrage, alors que l'informatique n'est pas sa tasse de thé.

A michel Veuillerot ex-Manager européen Information Technology and Telecom Architecture and Delivery Services chez Eastman Kodak, qui a relu attentivement la version précédente de l'ouvrage et testé tous les exemples.

Remerciements : (*diffusion de la connaissance*)
• A l'université de Tours qui supporte et donne accès à la partie Internet du package pédagogique à partir de sa rubrique "cours en ligne", à partir duquel ce document a été élaboré.

• Au club des développeurs francophones qui héberge gratuitement un site miroir du précédent et qui recommande le package pédagogique (http://rmdiscala.developpez.com/cours/) à ses visiteurs débutants.

Remerciements : (*anticipés*)

Aux lecteurs qui trouveront nécessairement encore des erreurs, des oublis, et autres imperfections et qui voudront bien les signaler à l'auteur afin d'améliorer le cours, **e-mail :** csharplivre@discala.net

Site de consultation et de téléchargement des autres ouvrages en pdf (Bases de l'informatique, Java 2) :
http://www.discala.net

Introduction à .Net

Une stratégie différente de répartition de l'information et de son traitement est proposée depuis 2001 par Microsoft, elle porte le nom de **.NET** (ou en anglais **dot net**). La conception de cette nouvelle architecture s'appuie sur quelques idées fondatrices que nous énonçons ci-dessous :

- Une disparition progressive des différences entre les applications et l'Internet, les serveurs ne fourniront plus seulement des pages HTML, mais des services à des applications distantes.
- Les informations au lieu de rester concentrées sur un seul serveur pourront être réparties sur plusieurs machines qui proposeront chacune un service adapté aux informations qu'elles détiennent.
- A la place d'une seule application, l'utilisateur aura accès à une fédération d'applications distantes ou locales capables de coopérer entre elles pour divers usages de traitement.
- L'utilisateur n'aurait plus la nécessité d'acheter un logiciel, il louerait plutôt les services d'une action spécifique.
- Le micro-ordinateur reste l'intermédiaire incontournable de cette stratégie, il dispose en plus de la capacité de terminal intelligent pour consulter et traiter les informations de l'utilisateur à travers Internet où qu'elles se trouvent.
- Offrir aux développeurs d'applications .NET un vaste ensemble de composants afin de faire de la programmation par composant unifiée au sens des protocoles (comme l'utilisation du protocole SOAP) et diversifiée quant aux lieux où se trouvent les composants.

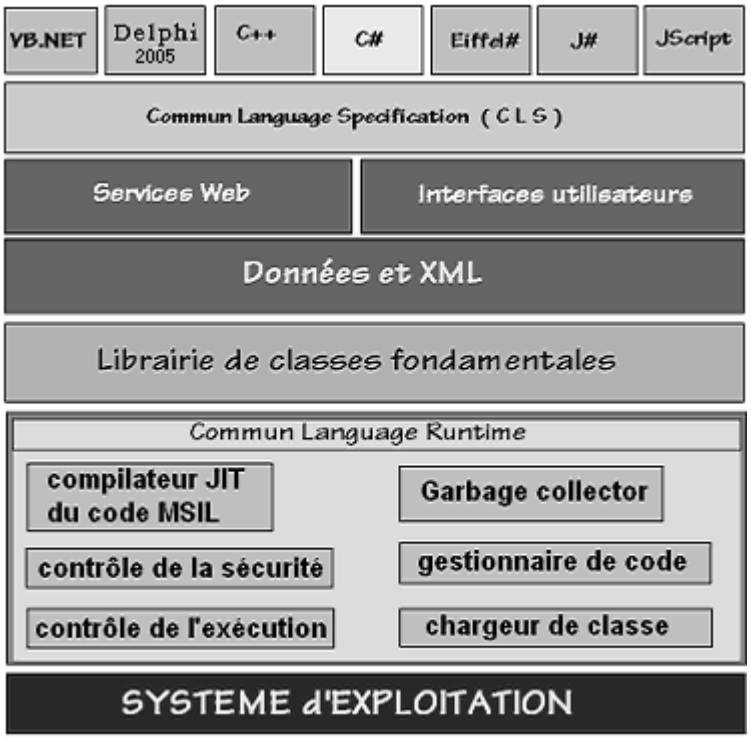
Afin de mettre en place cette nouvelle stratégie, microsoft procède par étapes. Les fondations de l'architecture **.NET** sont posées par l'introduction d'un environnement de développement et d'exécution des applications **.NET**. Cet environnement en version stabilisée depuis 2002 avec une révision majeure en 2005, porte la dénomination de **.NET Framework**, il est distribué gratuitement par microsoft sur toutes les versions de Windows (98, Me,..., Xp,...).

L'outil Visual Studio **.NET** contient l'environnement RAD de développement pour l'architecture **.NET**. Visual Studio **.NET** permet le développement d'applications classiques Windows ou Internet.

Les outils élémentaires

1. La plate forme .NET Framework

Elle comporte plusieurs couches les unes abstraites, les autres en code exécutable :



La première couche CLS est composée des spécifications communes à tous les langages qui veulent produire des applications **.NET** qui soient exécutables dans cet environnement et les langages eux-même. Le CLS est une sorte de sous-ensemble minimal de spécifications autorisant une interopérabilité complète entre tous les langages de **.NET** les règles minimales (il y en a en fait 41) sont :

- Les langages de **..NET** doivent savoir utiliser tous les composants du CLS
- Les langages de **.NET** peuvent construire de nouvelles classes, de nouveaux composants conformes au CLS

Le C# est le langage de base de **.NET**, il correspond à une synthèse entre Delphi et Java (le concepteur principal de **.NET**. et de C# est l'ancien chef de projet Turbo pascal puis Delphi de Borland).

Afin de rendre Visual Basic interopérable sur **.NET,** il a été entièrement reconstruit par microsoft et devient un langage orienté objet dénommé VB**.NET**.

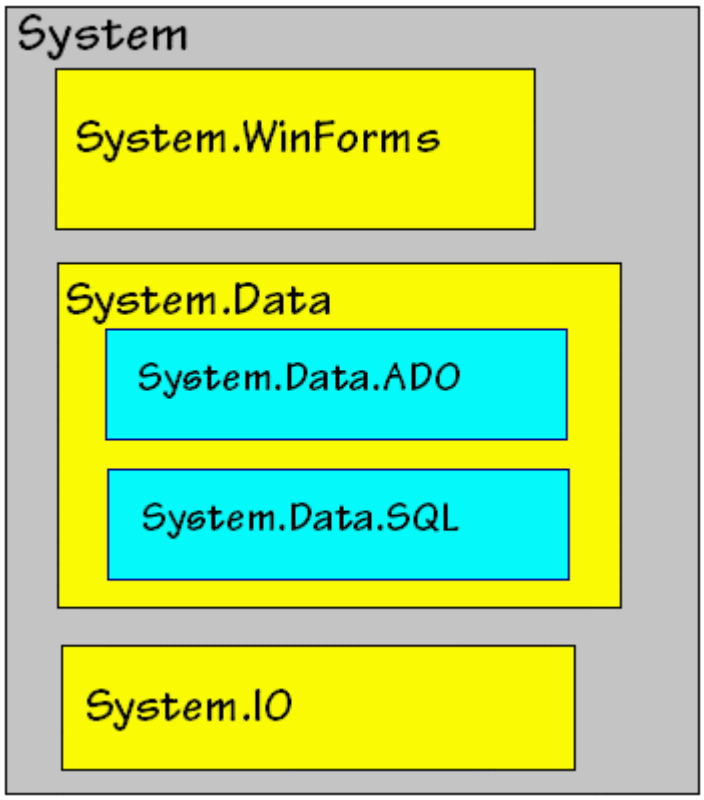
La seconde couche est un ensemble de composants graphiques disponibles dans Visual Studio **.NET** qui permettent de construire des interfaces homme-machine orientées Web (services Web) ou bien orientées applications classiques avec IHM.

Les données sont accédées dans le cas des services Web à travers les protocoles qui sont des standards de l'industrie : HTTP, XML et SOAP.

La troisième couche est constituée d'une vaste librairie de plusieurs centaines de classes :

Toutes ces classes sont accessibles telles quelles à tous les langages de **.NET** et cette librairie peut être étendue par adjonction de nouvelles classes. Cette librairie a la même fonction que la bibliothèque des classes de Java.

La librairie de classe de **.NET Framework** est organisée en nom d'espace hierarchisés, exemple ci-dessous de quelques espaces de nom de la hiérarchie System :



Un nom complet de classe comporte le "chemin" hiérarchique de son espace de nom et se termine par le nom de la classe exemples :

- 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**".

La quatrième couche forme l'environnement d'exécution commun (**CLR** ou **C**ommon **L**anguage **R**untime) de tous les programmes s'exécutant dans l'environnement .NET. Le **CLR** exécute un bytecode écrit dans un langage intermédiaire (**MSIL** ou **M**icro**S**oft **I**ntermediate **L**anguage)

Rappelons qu'un ordinateur ne sait exécuter que des programmes écrits en instructions machines compréhensibles par son processeur central. C# comme pascal, C etc... fait partie de la famille des langages évolués (ou langages de haut niveau) qui ne sont pas compréhensibles immédiatement par le processeur de l'ordinateur. Il est donc nécesaire d'effectuer une "**traduction**" d'un programme écrit en langage évolué afin que le processeur puisse l'exécuter.

Les deux voies utilisées pour exécuter un programme évolué sont la **compilation** ou l'**interprétation** :

Un compilateur du langage X pour un processeur P, est un logiciel qui traduit un programme source écrit en X en un programme cible écrit en instructions machines exécutables par le processeur P.

Un interpréteur du langage X pour le processeur P, est un logiciel qui ne produit pas de programme cible mais qui effectue lui-même immédiatement les opérations spécifiées par le programme source.

Un compromis assurant la portabilité d'un langage : une pseudo-machine

Lorsque le processeur P n'est pas une machine qui existe physiquement mais un logiciel simulant (ou interprétant) une machine on appelle cette machine pseudo-machine ou p-machine. Le programme source est alors traduit par le compilateur en instructions de la pseudo-machine et se dénomme pseudo-code. La p-machine standard peut ainsi être implantée dans n'importe quel ordinateur physique à travers un logiciel qui simule son comportement; un tel logiciel est appelé interpréteur de la p-machine.

La première p-machine d'un langage évolué a été construite pour le langage **pascal** assurant ainsi une large diffusion de ce langage et de sa version UCSD dans la mesure où le seul effort d'implementation pour un ordinateur donné était d'écrire l'interpréteur de p-machine pascal, le reste de l'environnement de développement (éditeurs, compilateurs,...) étant écrit en pascal était fourni et fonctionnait dès que la p-machine était opérationnelle sur la plate-forme cible.

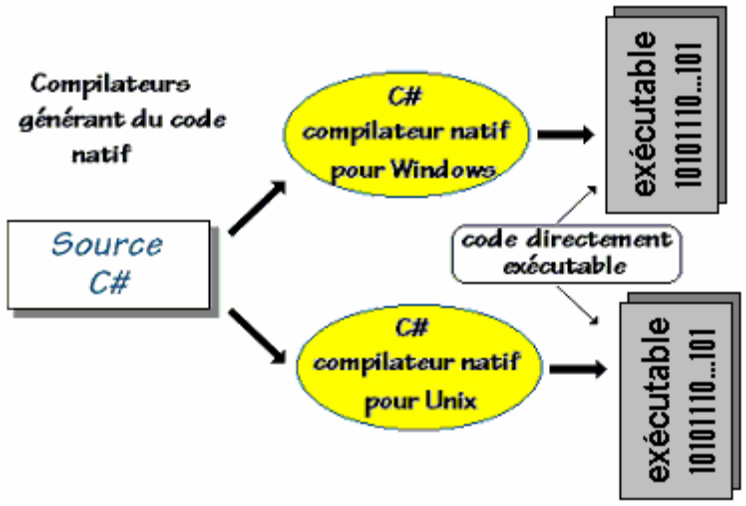
Donc dans le cas d'une p-machine le programme source est compilé, mais le programme cible est exécuté par l'interpréteur de la p-machine.

Beaucoup de langages possèdent pour une plate-forme fixée des interpréteurs ou des compilateurs, moins possèdent une p-machine, Java de Sun est l'un de ces langages. Tous les langages de la plateforme .NET fonctionnent selon ce principe, **C# conçu par microsoft en est le dernier**, un programme C# compilé en p-code, s'exécute sur la p-machine virtuelle incluse dans le CLR.

Nous décrivons ci-dessous le mode opératoire en C#.

Compilation native

La compilation native consiste en la traduction du source C# (éventuellement préalablement traduit instantanément en code intermédiare) en langage binaire exécutable sur la plate-forme concernée. Ce genre de compilation est équivalent à n'importe quelle compilation d'un langage dépendant de la plate-forme, l'**avantage est la rapidité d'exécution des instructions machines par le processeur central.** La stratégie de développement multi-plateforme de .Net, fait que Microsoft ne fournit pas pour l'instant, de compilateur C# natif, il faut aller voir sur le net les entreprises vendant ce type de produit.



Programe source C# : xxx.**cs**
Programe exécutable sous Windows : xxx.**exe** (code natif processeur)

Bytecode ou langage intermédiaire

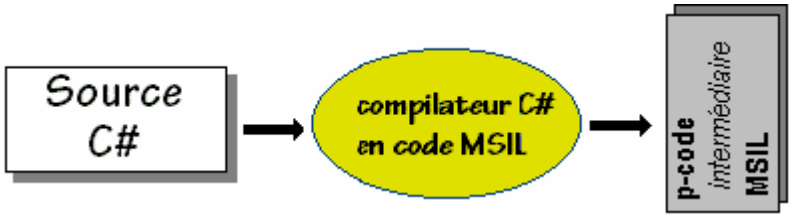
La compilation en bytecode (ou pseudo-code ou p-code ou code intermédiaire) est semblable à l'idée du p-code de N.Wirth pour obtenir un portage multi plate-formes du pascal. Le compilateur C# de **.NET Framework** traduit le programme source xxx.**cs** en un code intermédiaire indépendant de toute machine physique et non exécutable directement, le fichier obtenu se dénomme PE (portable executable) et prend la forme : xxx.**exe**.

Seule une p-machine (dénommée **machine virtuelle .NET**) est capable d'exécuter ce bytecode. Le bytecode est aussi dénommé **MSIL**. En fait le bytecode MSIL est pris en charge par le CLR et n'est pas interprété par celui-ci mais traduit en code natif du processeur et exécuté par le processeur sous contrôle du CLR..

ATTENTION

Bien que se terminant par le suffixe **exe**, un programme issu d'une compilation sous **.NET** n'est pas un exécutable en code natif, mais un bytecode en **MSIL**; ce qui veut dire que **vous ne pourrez pas faire exécuter directement** sur un ordinateur qui n'aurait pas la machine virtuelle **.NET,** un programme **PE** "xxx.**exe**" ainsi construit.

Ci-dessous le schéma d'un programme source ***Exemple.cs*** traduit par le compilateur C# sous **.NET** en un programme cible écrit en bytecode nommé ***Exemple.exe***

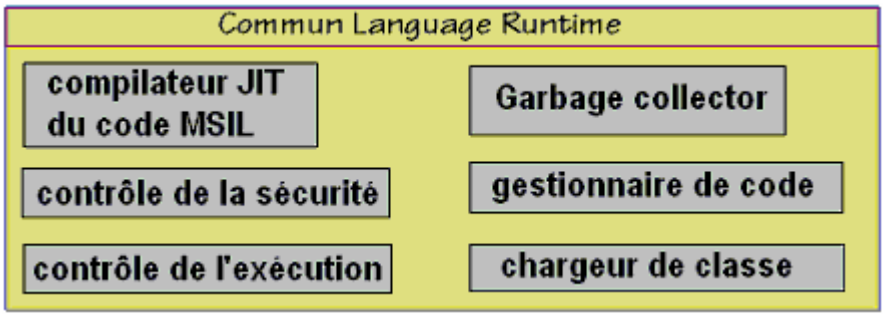


Programe source C# : Exemple.**cs**
Programe exécutable sous **.NET** : Exemple.**exe** (code portable IL)

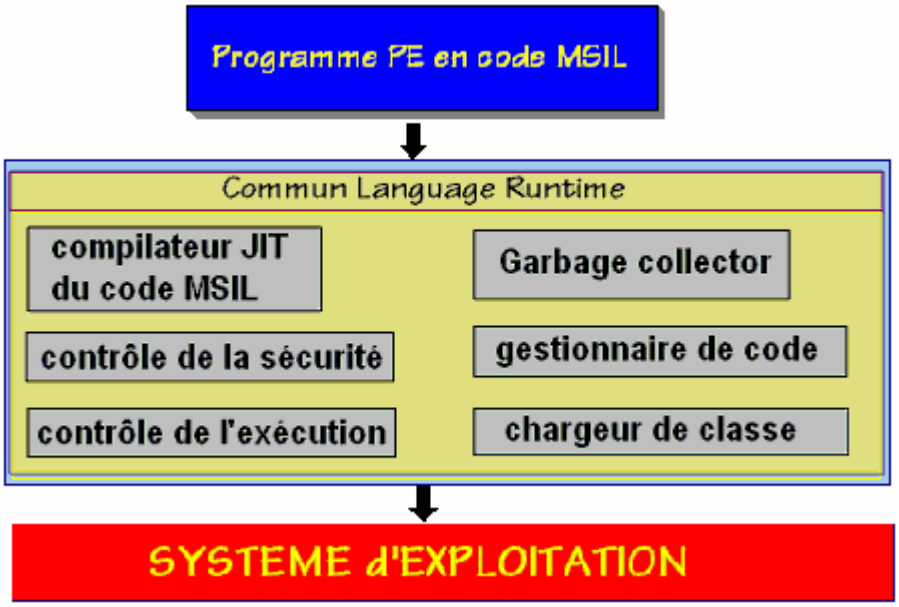
2. L'environnement d'exécution du CLR

Rappelons que le **CLR** (Common Language Runtime) est un environnement complet d'exécution semblable au JRE de Sun pour Java, il est indépendant de l'architecture machine sous-jacente. Le **CLR** prend en charge essentiellement :

- le chargement des classes,
- les vérifications de types,
- la gestion de la mémoire, des exceptions, de la sécurité,
- la traduction à la volée du code MSIL en code natif (compilateur interne JIT),
- à travers le CTS (Common Type System) qui implémente le CLS (Common Language Specification), le CLR assure la sécurité de compatibilité des types connus mais syntaxiquement différents selon les langages utilisés.



Une fois le programme source C# traduit en bytecode **MSIL**, la machine virtuelle du **CLR** se charge de l'exécuter sur la machine physique à travers son système d'exploitation (Windows, Unix,...)



Le CLR intégré dans l'environnement .NET est distribué gratuitement.

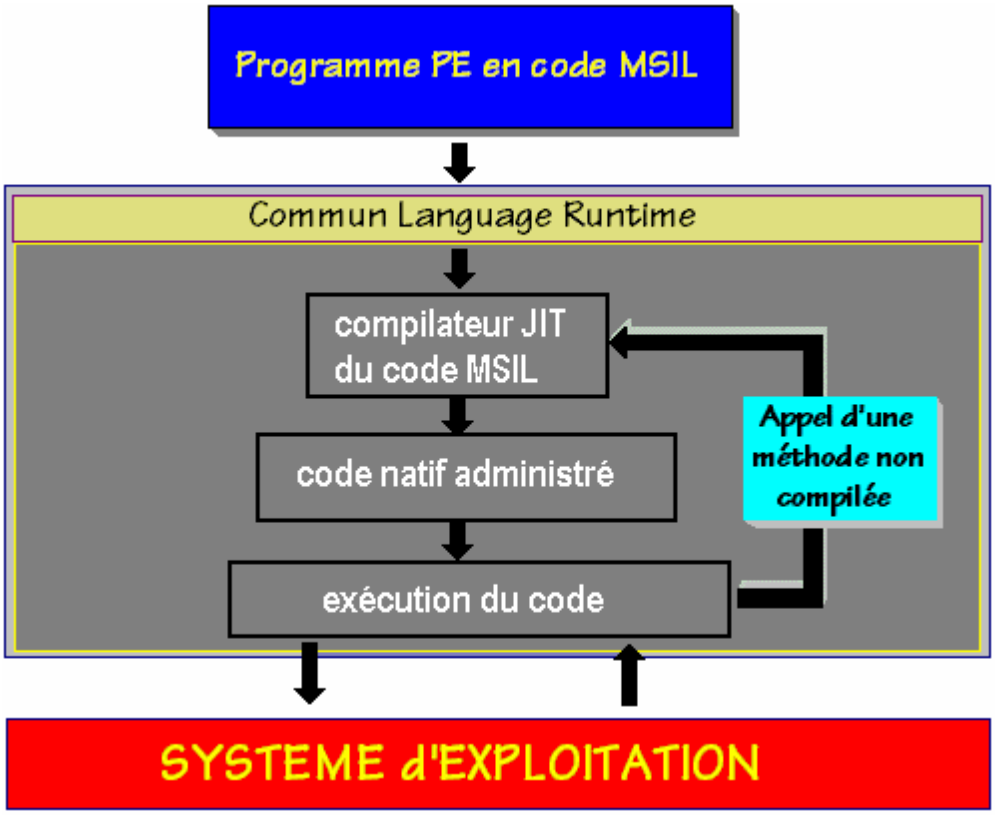
La compilation JIT progressive

L'interprétation et l'exécution du bytecode ligne par ligne pourrait prendre beaucoup de temps et cela a été semble-t-il le souci de microsoft qui a adopté une stratégie d'optimisation de la vitesse d'exécution du code MSIL en utilisant la technique **Just-in-time.**

JIT *(Just-in-time)* est une technique de **traduction dynamique durant l'interprétation**. La machine virtuelle CLR contient un compilateur optimiseur qui **recompile localement le bytecode MSIL** afin de n'avoir plus qu'à faire exécuter des instructions machines de base. Le compilateur **JIT** du CLR compile une méthode en code natif dès qu'elle est appelée dans le code MSIL, le processus recommence à chaque fois qu'un appel de méthode a lieu sur une méthode non déjà compilée en code natif.

On peut mentalement considérer qu'avec cette technique vous obtenez un programme C# cible compilé en deux passages :

- le premier passage est dû à l'utilisation du compilateur C# produisant exécutable portable (**PE**) en bytecode **MSIL**,
- le second passage étant le compilateur **JIT** lui-même qui optimise et traduit localement à la volée et à chaque appel de méthode, le bytecode **MSIL** en instructions du processeur de la plate-forme. Ce qui donne au bout d'un temps très bref, un code totalement traduit en instruction du processeur de la plateforme, selon le schéma ci-après :



La compilation AOT

Toujours à des fins d'optimisation de la vitesse d'exécution du code MSIL, la technique **AOT Ahead-Of-Time** est employée dans les versions récentes de .Net depuis 2005.

AOT *(ahead-of-time)* est une technique de **compilation locale de tout le bytecode MSIL** avant exécution (semblable à la compilation native). Le compilateur **AOT** du CLR compile, avant une quelconque exécution et en une seule fois, toutes les lignes de code MSIL et génère des images d'exécutables à destination du CLR.

Les éléments de base

Tout est objet dans C#, en outre C# est un langage fortement typé. Comme en Delphi et en Java vous devez déclarer un objet C# ou une variable C# avec son type avant de l'utiliser. C# dispose de **types valeurs intrinsèques** qui sont définis à partir des types de base du CLS (Common Language Specification).

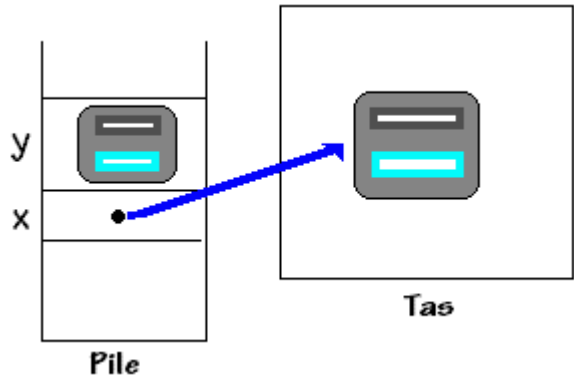
1. Les types valeurs du CLS dans .NET Framework

Struct

Les classes encapsulant les types élémentaires dans **.NET Framework** sont des classes de type valeur du genre **structures**. Dans le CLS une classe de type valeur est telle que les allocations d'objets de cette classe se font directement dans la pile et non dans le tas, il n'y a donc pas de référence pour un objet de type valeur et lorsqu'un objet de type valeur est passé comme paramètre il est **passé par valeur**.

Dans **.NET Framework** les classes-structures de type valeur sont déclarées comme structures et ne sont pas dérivables, les classes de type référence sont déclarées comme des classes classiques et sont dérivables.

Afin d'éclairer le lecteur prenons par exemple un objet x instancié à partir d'une classe de type référence et un objet y instancié à partir d'un classe de type valeur contenant les mêmes membres que la classe par référence. Ci-dessous le schéma d'allocation de chacun des deux objets :



En C# on aurait le genre de syntaxe suivant :

Déclaration de classe-structure :	**instanciation :**

struct StructAmoi **{**
int b;
void meth(**int** a){
b = 1000+a;
}
}
	StructAmoi y = **new** StructAmoi ( ) ;

Déclaration de classe :	**instanciation :**

class ClassAmoi **{**
int b;
void meth(**int** a) {
b = 1000+a;
}
}
	ClassAmoi x = **new** ClassAmoi ( ) ;

Les classes-structures de type valeur peuvent comme les autres classes posséder un constructeur explicite, qui comme pour toute classe C# doit porter le même nom que celui de la classe-structure.

Exemple ci-desssous d'une classe-structure dénommée Menulang:

```
public struct Menulang
{
  public String MenuTexte;
  public String Filtre;
  public Menulang(String M, String s)
  {
     MenuTexte = M;
     Filtre = s;
  }
}
```

On instancie alors un objet de type valeur comme un objet de type référence.

En reprenant l'exemple de la classe précédente on instancie et on utilise un objet Rec :

```
Menulang Rec = new Menulang ( Nomlang , FiltreLang );
Rec.MenuTexte = "Entrez" ;
Rec.Filtre = "*.ent" ;
```

Classe-structure	*intervalle de variation*	*nombre de bits*
Boolean	**false , true**	1 bit

SByte	octet signé -128 ... +127	8 bits
Byte	octet non signé 0 ... 255	8 bits
Char	caractères unicode (valeurs de 0 à 65535)	16 bits
Double	Virgule flottante double précision ~ 15 décimales	64 bits
Single	Virgule flottante simple précision ~ 7 décimales	32 bits
Int16	entier signé court [-2^{15}... $+2^{15}-1$]	16 bits
Int32	entier signé [-2^{31}... $+2^{31}-1$]	32 bits
Int64	entier signé long [-2^{63}... $+2^{63}-1$]	64 bits
UInt16	entier non signé court $0\ldots2^{16}-1$	16 bits
UInt32	entier non signé $0\ldots2^{32}-1$	32 bits
UInt64	entier non signé long $0\ldots2^{64}-1$	64 bits
Decimal	réeel = entier* 10^n (au maximum 28 décimales exactes)	128 bits

Compatibilité des types de .NET Framework

Le type **System.Int32** est le type valeur entier signé sur 32 bits dans le CLS.
Voici selon 4 langages de **.NET Framework** (VB, C#, C++, J#) la déclaration syntaxique du type **Int32** :

[*Visual Basic*]
Public Structure Int32
Implements IComparable, IFormattable, IConvertible

[*C#*]
public struct Int32 : IComparable, IFormattable, IConvertible

[*C++*]
public __value struct Int32 : **public** IComparable, IFormattable, IConvertible

[*J#*]
public class Int32 **extends** System.ValueType **implements** System.IComparable, System.IFormattable, System.IConvertible

Les trois premières déclarations comportent syntaxiquement le mot clef **struct ou Structure**

indiquant le mode de gestion **par valeur** donc sur la pile des objets de ce type. La dernière déclaration en J# compatible syntaxiquement avec Java, utilise une classe qui par contre gère ses objets **par référence** dans le tas. C'est le CLR qui va se charger de maintenir une cohérence interne entre ces différentes variantes; ici on peut raisonnablement supposer que grâce au mécanisme d'emboîtage (Boxing) le CLR allouera un objet par référence encapsulant l'objet par valeur, mais cet objet encapsulé sera marqué comme objet-valeur.

enum

Un type **enum** est un type valeur qui permet de déclarer un ensemble de constantes de base comme en pascal. En C#, chaque énumération de type **enum,** possède un type sous-jacent, qui peut être de n'importe quel type entier : byte, sbyte, short, ushort, int, uint, long ou ulong.

Le type **int** est le type sous-jacent par défaut des éléments de l'énumération. Par défaut, le premier énumérateur a la valeur 0, et l'énumérateur de rang **n** a la valeur **n-1**.

Soit par exemple un type énuméré jour :

```
enum jour { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche}
par défaut :  rang de lundi=0, rang de mardi=1, ... , rang de dimanche=6
```

1°) Il est possible de déclarer classiquement une variable du type jour comme un objet de type jour, de l'instancier et de l'affecter :

```
jour unJour = new jour ( );
unJour = jour.lundi ;
int rang = (int)unJour; // rang de la constante dans le type énuméré
System.Console.WriteLine("unJour = "+unJour.ToString()+" , place = '+rang);

Résultat de ces 3 lignes de code affiché sur la console :
unJour = lundi  ,  place = 0
```

2°) Il est possible de déclarer d'une manière plus courte la même variable du type jour et de l'affecter :

```
jour unJour ;
unJour = jour.lundi ;

int rang = (int)unJour;
System.Console.WriteLine("unJour = "+unJour.ToString()+" , place = '+rang);

Résultat de ces 3 lignes de code affiché sur la console :
unJour = lundi  ,  place = 0
```

Remarque
C# accepte que des énumérations aient des noms de constantes d'énumérations identiques :

```
enum jour { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche}
enum weekEnd { vendredi, samedi, dimanche}
```

Dans cette éventualité faire attention, la comparaison de deux variables de deux types différents, affectées chacune à une valeur de constante identique dans les deux types, ne conduit pas à l'égalité de ces variables (c'est en fait le rang dans le type énuméré qui est testé). L'exemple ci-dessous illustre cette remarque :

```
enum jour { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche}
enum weekEnd { vendredi, samedi, dimanche}
jour unJour ;
weekEnd  repos ;
unJour = jour.samedi ;
repos = weekEnd.samedi;
if ( (jour)repos == unJour ) // il faut transtyper l'un des deux si l'on veut les comparer
   System.Console.WriteLine("Le même jour");
else
   System.Console.WriteLine("Jours différents");
```

Résultat de ces lignes de code affiché sur la console :
Jours différents

2. Syntaxe des types valeurs de C# et transtypage

Les types servent à déterminer la nature du contenu d'une variable, du résultat d'une opération, d'un retour de résultat de fonction.

Ci-dessous le tableau de correspondance syntaxique entre les types élémentaires du C# et les classes de .NET Framework (table appelée aussi, table des alias) :

Types valeurs C#	*Classe-structure de .NET Framework*	*nombre de bits*
bool	Boolean	1 bit
sbyte	SByte	8 bits
byte	Byte	8 bits

char	Char	16 bits
double	Double	64 bits
float	Single	32 bits
short	Int16	16 bits
int	Int32	32 bits
long	Int64	64 bits
ushort	UInt16	16 bits
uint	UInt32	32 bits
ulong	UInt64	64 bits
decimal	Decimal	128 bits

Rappelons qu'en C# toute variable qui sert de conteneur à une valeur d'un type élémentaire précis doit préalablement avoir été déclarée sous ce type.

Remarque importante

Une variable de type élémentaire en C# est (pour des raisons de compatibilité CLS) automatiquement un objet de type valeur (*Par exemple une variable de type **float** peut être considérée comme un objet de classe **Single***).**

Il est possible d'indiquer au compilateur le type d'une valeur numérique en utilisant un suffixe :

- **l** ou **L** pour désigner un entier du type long
- **f** ou **F** pour désigner un réel du type float
- **d** ou **D** pour désigner un réel du type double
- **m** ou **M** pour désigner un réel du type decimal

Exemples :

45**l** ou 45**L** **représente la valeur 45 en entier signé sur 64 bits.**
45**f** ou 45**F** **représente la valeur 45 en virgule flottante simple précision sur 32 bits.**
45**d** ou 45**D** **représente la valeur 45 en virgule flottante double précision sur 64 bits.**

5.27e-2**f** ou 5.27e-2**F** **représente la valeur 0.0527 en virgule flottante simple précision sur 32 bits.**

Transtypage opérateur ( )

Les conversions de type en C# sont identiques pour les types numériques aux conversions utilisées dans un langage fortement typé comme Delphi par exemple. Toutefois C# pratique la **conversion implicite** lorsque celle-ci est possible. Si vous voulez malgré tout, convertir explicitement une valeur immédiate ou une valeur contenue dans une variable il faut utiliser l'opérateur de transtypage noté **( )**. Nous nous sommes déjà servi de la fonctionnalité de transtypage explicite au paragraphe précédent dans l'instruction : **int** rang = (**int**)unJour; et dans l'instruction **if** ((jour)repos == unJour)...

Transtypage implicite en C# :

- **int** n = 1234;
- **float** x1 = n ;
- **double** x2 = n ;
- **double** x3 = x1 ;
- **long** p = n ;
  .....

Transtypage explicite en C# :

int x;
x = **(int)** y ; signifie que vous demandez de **transtyper** la valeur contenue dans la variable y en **un entier signé 32 bits** avant de la mettre dans la variable x.

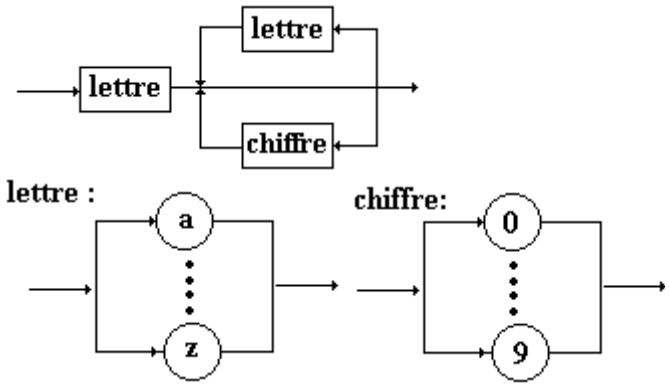
- Tous les types élémentaires peuvent être transtypés à l'exception du type **bool** qui ne peut pas être converti en un autre type (différence avec le C).
- Les conversions **peuvent être restrictives** quant au résultat; par exemple le transtypage du réel 5.27e-2 en entier (x = **(int)**5.27e-2) mettra l'entier zéro dans x.

3. Variables, valeurs, constantes en C#

Comme en Java, une variable C# peut contenir soit une valeur d'un type élémentaire, soit une référence à un objet. Les variables jouent le même rôle que dans les langages de programmation classiques impératifs, leur visibilité est étudiée dans le prochain chapitre.

Les identificateurs de variables en C# se décrivent comme ceux de tous les langages de programmation :

Identificateur C# :



Attention C# fait une différence entre majuscules et minuscules, c'est à dire que la variable **BonJour** n'est pas la même que la variable **bonjour** ou encore la variable **Bonjour**. En plus des lettres, les caractères suivants sont autorisés pour construire une identificateur C# : "$" , "_" , "µ" et les lettres accentuées.

Exemples de déclaration de variables :

int Bonjour ; **int** µEnumération_fin$;
float Valeur ;
char UnCar ;
bool Test ;

etc ...

Exemples d'affectation de valeurs à ces variables :

Affectation	*Déclaration avec initialisation*
Bonjour = 2587 ; Valeur = -123.5687 UnCar = 'K' ; Test = false ;	**int** Bonjour = 2587 ; **float** Valeur = -123.5687 **char** UnCar = 'K' ; **bool** Test = false ;

Exemple avec transtypage :
int Valeur ;
char car = '8' ;
Valeur = (**int**)car - (**int**)'0';

fonctionnement de l'exemple :

Lorsque la variable car est l'un des caractères '0', '1', ... ,'9', la variable Valeur est égale à la valeur numérique associée (il s'agit d'une conversion car = '0' ---> Valeur = 0, car = '1' ---> Valeur = 1, ... , car = '9' ---> Valeur = 9).

Les constantes en C#

C# dispose de deux mots clefs pour qualifier des variables dont le contenu ne peut pas être modifié : **const** et **readonly** sont des qualificateurs de déclaration qui se rajoutent devant les autres qualificateurs de déclaration..

- Les constantes qualifiées par **const** doivent être initialisées lors de leur déclaration. Une variable membre de classe ou une variable locale à une méthode peut être qualifiée en constante **const**. Lorsque de telles variables sont déclarées comme variables membre de classe, elles sont considérées comme des **variables de classe statiques** :

- **const int** x ; *// erreur , le compilateur n'accepte pas une constante non initialisée.*
- **const int** x = 1000 ; *// x est déclarée comme constante entière initialisée à 1000.*
- x = 8 ; *<------ provoquera une erreur de compilation interdisant la modification de la valeur de x.*

- Les constantes qualifiées par **readonly** sont uniquement des variables membre de classes, elles peuvent être initialisées dans le constructeur de la classe (et uniquement dans le constructeur) :

- **readonly int** x ; *// correct.*
- **readonly int** x = 100 ; *// correct.*

-Rappelons enfin pour mémoire les constantes de base d'un type énuméré (cf. enum)

Base de représentation des entiers

C# peut représenter les entiers dans 2 bases de numération différentes : décimale (base 10), hexadécimale (base 16). La détermination de la base de représentation d'une valeur est d'ordre syntaxique grâce à un préfixe :

- **pas de préfixe ----> base = 10 *décimal*.**
- **préfixe 0x ----> base = 16 *hexadécimal***

Les opérateurs

1. Priorité d'opérateurs en C#

Les opérateurs du C# sont très semblables à ceux de Java et donc de C++, ils sont détaillés par famille, plus loin. Ils sont utilisés comme dans tous les langages impératifs pour manipuler, séparer, comparer ou stocker des valeurs. Les opérateurs ont soit un seul opérande, soit deux opérandes, il n'existe en C# qu'un seul opérateur à trois opérandes (comme en Java) l'opérateur conditionnel " **? :** ".

Dans le tableau ci-dessous les opérateurs de C# sont classés par ordre de priorité croissante (0 est le plus haut niveau, 13 le plus bas niveau). Ceci sert lorsqu'une expression contient plusieurs opérateurs à **indiquer l'ordre dans lequel s'effectueront les opérations**.

- Par exemple sur les entiers l'expression 2+3*4 vaut 14 car l'opérateur * est plus prioritaire que l'opérateur +, donc l'opérateur * est effectué en premier.
- Lorsqu'une expression contient des opérateurs de **même priorité alors C# effectue les évaluations de gauche à droite**. Par exemple l'expression 12/3*2 vaut 8 car C# effectue le parenthésage automatique de gauche à droite ((12/3)*2).

Tableau général de toutes les priorités

priorité	tous les opérateurs de C#
0	( ) [ ] . **new**
1	! ~ ++ --
2	* / %
3	+ -
4	<< >>
5	< <= > >= **is**
6	= = !=
7	&

8	^
9	|
10	&&
11	||
12	? :
13	= *= /= %= += -= ^= &= <<= >>= >>>= |=

2. Les opérateurs arithmétiques en C#

Les opérateurs d'affectation seront mentionnés plus loin comme cas particulier de l'instruction d'affectation.

opérateurs travaillant avec des opérandes à valeur immédiate ou variable

Opérateur	priorité	action	exemples
+	1	signe positif	+a; +(a-b); +7 (unaire)
-	1	signe négatif	-a; -(a-b); -7 (unaire)
*	2	multiplication	5*4; 12.7*(-8.31); 5*2.6
/	2	division	5 / 2; 5.0 / 2; 5.0 / 2.0
%	2	reste	5 % 2; 5.0 %2; 5.0 % 2.0
+	3	addition	a+b; -8.53 + 10; 2+3
-	3	soustraction	a-b; -8.53 - 10; 2-3

Ces opérateurs sont binaires (à deux opérandes) exceptés les opérateurs de signe positif ou négatif. Ils travaillent tous avec des opérandes de types entiers ou réels. Le résultat de l'opération est converti automatiquement en valeur du type des opérandes.

L'opérateur " **%** " de reste n'est intéressant que pour des calculs sur les entiers longs, courts, signés ou non signés : il renvoie le reste de la division euclidienne de 2 entiers.

Exemples d'utilisation de l'opérateur de division selon les types des opérandes et du résultat :

programme C#	***résultat obtenu***	***commentaire***
int x = 5 , y ;	**x = 5 , y =???**	*déclaration*
float a , b = 5 ;	**b = 5 , a =???**	*déclaration*
y = x / 2 ;	**y = 2** *// type int*	**int** x et **int** 2 *résultat* **: int**
y = b / 2 ;	***erreur de conversion : interdit***	*conversion implicite impossible (**float** b --> **int** y)*
y = b / 2.0 ;	***erreur de conversion: interdit***	*conversion implicite impossible (**float** b --> **int** y)*
a = b / 2 ;	**a = 2.5** *// type float*	**float** b et **int** 2 *résultat* **: float**
a = x / 2 ;	**a = 2.0** *// type float*	**int** x et **int** 2 *résultat* **: int** *conversion automatique* ***int** 2 --> **float** 2.0*
a = x / 2f ;	**a = 2.5** *// type float*	**int** x et **float** 2f *résultat* **: float**

Pour l'instruction précédente " **y = b / 2** " engendrant une erreur de conversion voici deux corrections possibles utilisant le transtypage explicite :

y = (**int**)b / 2 ; *// b est converti en **int** avant la division qui s'effectue sur deux **int**.*
y = (**int**)(b / 2) ; *// c'est le résultat de la division qui est converti en **int**.*

opérateurs travaillant avec une unique variable comme opérande

Opérateur	**priorité**	**action**	**exemples**
++	1	post ou pré incrémentation : incrémente de 1 son opérande numérique : short, int, long, char, float, double.	++a; a++; (unaire)
--	1	post ou pré décrémentation : décrémente de 1 son opérande numérique : short, int, long, char, float, double.	--a; a--; (unaire)

L'objectif de ces opérateurs est l'optimisation de la vitesse d'exécution du bytecode MSIL dans le CLR (cette optimisation n'est pas effective dans le version actuelle du MSIL) mais surtout la reprise syntaxique aisée de code source Java et C++.

post-incrémentation : k++

la valeur de k est d'abord utilisée telle quelle dans l'instruction, puis elle est augmentée de un à la fin. Etudiez bien les exemples ci-après qui vont vous permettre de bien comprendre le fonctionnement de cet opérateur.

Nous avons mis à côté de l'instruction C# les résultats des contenus des variables après exécution de l'instruction de déclaration et de la post incrémentation.

Exemple 1 :

int k = 5 , n ; **n = k++ ;**	**n = 5**	**k = 6**

Exemple 2 :

int k = 5 , n ; **n = k++ - k ;**	**n = -1**	**k = 6**

Dans l'instruction k++ - k nous avons le calcul suivant : la valeur de k (k=5) est utilisée comme premier opérande de la soustraction, puis elle est incrémentée (k=6), la nouvelle valeur de k est maintenant utilisée comme second opérande de la soustraction ce qui revient à calculer n = 5-6 et donne n = -1 et k = 6.

Exemple 3 :

int k = 5 , n ; **n = k - k++ ;**	**n = 0**	**k = 6**

Dans l'instruction k - k++ nous avons le calcul suivant : la valeur de k (k=5) est utilisée comme premier opérande de la soustraction, le second opérande de la soustraction est k++ c'est la valeur actuelle de k qui est utilisée (k=5) avant incrémentation de k, ce qui revient à calculer n = 5-5 et donne n = 0 et k = 6.

Exemple 4 : Utilisation de l'opérateur de post-incrémentation en combinaison avec un autre opérateur unaire.
int nbr1, z , t , u , v ;

nbr1 = 10 ; **v = nbr1++**	**v = 10**	**nbr1 = 11**
nbr1 = 10 ; **z = ~ nbr1 ;**	**z = -11**	**nbr1 = 10**
nbr1 = 10 ; **t = ~ nbr1 ++ ;**	**t = -11**	**nbr1 = 11**
nbr1 = 10 ; **u = ~ (nbr1 ++) ;**	**u = -11**	**nbr1 = 11**

La notation " **(~ nbr1) ++** *" est refusée par C# comme pour Java.*

Remarquons que les expressions **"~nbr1 ++ "** et **"~ (nbr1 ++)"** produisent les mêmes effets, ce qui est logique puisque lorsque deux opérateurs (ici ~ et ++) ont la même priorité, l'évaluation a lieu de gauche à droite.

pré-incrémentation : ++k

la valeur de k est d'abord augmentée de un ensuite utilisée dans l'instruction.

Exemple1 :
int k = 5 , n ;

n = ++k ;	**n = 6**	**k = 6**

Exemple 2 :

int k = 5 , n ; **n = ++k - k ;**	**n = 0**	**k = 6**

Dans l'instruction ++k - k nous avons le calcul suivant : le premier opérande de la soustraction étant ++k c'est donc la valeur incrémentée de k (k=6) qui est utilisée, cette même valeur sert de second opérande à la soustraction ce qui revient à calculer n = 6-6 et donne n = 0 et k = 6.

Exemple 3 :

int k = 5 , n ; **n = k - ++k ;**	**n = -1**	**k = 6**

Dans l'instruction k - ++k nous avons le calcul suivant : le premier opérande de la soustraction est k (k=5), le second opérande de la soustraction est ++k, k est immédiatement incrémenté (k=6) et c'est sa nouvelle valeur incrémentée qui est utilisée, ce qui revient à calculer n = 5-6 et donne n = -1 et k = 6.

post-décrémentation : k--

la valeur de k est d'abord utilisée telle quelle dans l'instruction, puis elle est diminuée de un à la fin.

Exemple1 :
int k = 5 , n ;

n = k-- ;	**n = 5**	**k = 4**

pré-décrémentation : --k
la valeur de k est d'abord diminuée de un, puis utilisée avec sa nouvelle valeur.

Exemple1 :
int k = 5 , n ;

n = --k ;	**n = 4**	**k = 4**

Reprenez avec l'opérateur - - des exemples semblables à ceux fournis pour l'opérateur ++ afin d'étudier le fonctionnement de cet opérateur (étudiez (- -k - k) et (k - - -k)).

3. Opérateurs de comparaison

Ces opérateurs employés dans une expression renvoient un résultat de type booléen (**false** ou **true**). Nous en donnons la liste sans autre commentaire car ils sont strictement identiques à tous les opérateurs classiques de comparaison de n'importe quel langage algorithmique (C, pascal, etc...). Ce sont des opérateurs à deux opérandes.

Opérateur	priorité	action	exemples
<	5	strictement inférieur	5 < 2 ; x+1 < 3 ; y-2 < x*4
<=	5	inférieur ou égal	-5 <= 2 ; x+1 <= 3 ; etc...
>	5	strictement supérieur	5 > 2 ; x+1 > 3 ; etc...
>=	5	supérieur ou égal	5 >= 2 ; etc...
= =	6	égal	5 = = 2 ; x+1 = = 3 ; etc...
!=	6	différent	5 != 2 ; x+1 != 3 ; etc...
is	5	Teste le type de l'objet	X **is** int ; **if** (x **is** Object) etc...

4. Opérateurs booléens

Ce sont les opérateurs classiques de l'algèbre de boole **{** { **V**, **F** }, ! , & , | **}** où { **V**, **F** } représente l'ensemble {**V**rai,**F**aux}. Les connecteurs logiques ont pour syntaxe en C# : **! , & , |, ^**:

& : { **V**, **F** } x { **V**, **F** } → { **V**, **F** } *(opérateur **binaire** qui se lit " et ")*

| : { **V**, **F** } x { **V**, **F** } → { **V**, **F** } *(opérateur **binaire** qui se lit " ou ")*

^ : { **V**, **F** } x { **V**, **F** } → { **V**, **F** } *(opérateur **binaire** qui se lit " ou exclusif ")*

! : { **V**, **F** } →{ **V**, **F** } *(opérateur **unaire** qui se lit " non ")*

Table de vérité des opérateurs (p et q étant des expressions booléennes)

p	q	! p	p & q	P ^ q	p | q
V	V	F	V	F	V
V	F	F	F	V	V
F	V	V	F	V	V
F	F	V	F	F	F

Remarque :

$\forall p \in \{ \mathbf{V}, \mathbf{F} \}$, $\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, p **&**q est toujours évalué en entier *(p et q sont toujours évalués).*
$\forall p \in \{ \mathbf{V}, \mathbf{F} \}$, $\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, p |q est toujours évalué en entier *(p et q sont toujours évalués).*

C# dispose de 2 clones des opérateurs binaires **&** et | . Ce sont les opérateurs **&&** et || qui se différentient de leurs originaux **&** et | par leur mode d'exécution optimisé (application de théorèmes de l'algèbre de boole) :

L'opérateur et optimisé : &&

Théorème
$\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, **F** **&**q = **F**

Donc si p est faux (p = **F**) , il est inutile d'évaluer q car l'expression p **&**q est fausse (p **&**q = **F**), comme l'opérateur **&** évalue toujours l'expression q, C# à des fins d'optimisation de la vitesse d'exécution du bytecode MSIL dans le CLR , propose un opérateur « et » noté **&&** qui a **la même table de vérité** que l'opérateur **&** mais qui applique ce théorème.

$\forall p \in \{ \mathbf{V}, \mathbf{F} \}$, $\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, p **&&**q = p **&**q
Mais dans p**&&**q , *q n'est évalué que si p* = ***V***.

L'opérateur ou optimisé : | |

Théorème
$\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, **V** |q = **V**

Donc si p est vrai (p = **V**), il est inutile d'évaluer q car l'expression p |q est vraie (p |q = **V**), comme l'opérateur | évalue toujours l'expression q, C# à des fins d'optimisation de la vitesse d'exécution du bytecode dans la machine virtuelle C#, propose un opérateur ou noté || qui

applique ce théorème et qui a la même table de vérité que l'opérateur | .

> ∀p ∈ { **V**, **F** } , ∀q∈ { **V**, **F** } , p ||q = p |q
> Mais dans p||q , *q n'est évalué que si p* = ***F***.

En résumé:

Opérateur	priorité	action	exemples
!	1	**non** booléen	! (5 < 2) ; !(x+1 < 3) ; etc...
&	7	**et** booléen complet	(5 = = 2) & (x+1 < 3) ; etc...
|	9	**ou** booléen complet	(5 != 2) | (x+1 >= 3) ; etc...
&&	10	**et** booléen optimisé	(5 = = 2) && (x+1 < 3) ; etc...
||	11	**ou** booléen optimisé	(5 != 2) || (x+1 >= 3) ; etc...

Nous allons voir ci-après une autre utilisation des opérateurs **&** et **|** sur des variables ou des valeurs immédiates en tant qu'opérateur bit-level.

5. Opérateurs bits level

Ce sont des opérateurs de bas niveau en C# dont les opérandes sont exclusivement l'un des types entiers ou caractère de C# (**short**, **int**, **long**, **char**, **byte**). Ils permettent de manipuler directement les bits du mot mémoire associé à la donnée.

Opérateur	priorité	action	exemples
~	1	**complémente** les bits	~a; ~(a-b); ~7 (unaire)
<<	4	**décalage** gauche	x << 3 ; (a+2) << k ; -5 << 2 ;
>>	4	**décalage** droite avec signe	x >> 3 ; (a+2) >> k ; -5 >> 2 ;
&	7	**et** booléen bit à bit	x & 3 ; (a+2) & k ; -5 & 2 ;
^	8	**ou** exclusif xor bit à bit	x ^ 3 ; (a+2) ^ k ; -5 ^ 2 ;
|	9	**ou** booléen bit à bit	x | 3 ; (a+2) | k ; -5 | 2 ;

Les tables de vérités des opérateurs "&", " | " et celle du ou exclusif " ^ " au niveau du bit sont identiques aux tables de verité booléennes (seule la valeur des constantes **V** et **F** change, **V** est remplacé par le bit **1** et **F** par le bit **0**) .

Table de vérité des opérateurs bit level

p	q	~ p	p & q	p | q	p ^ q
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

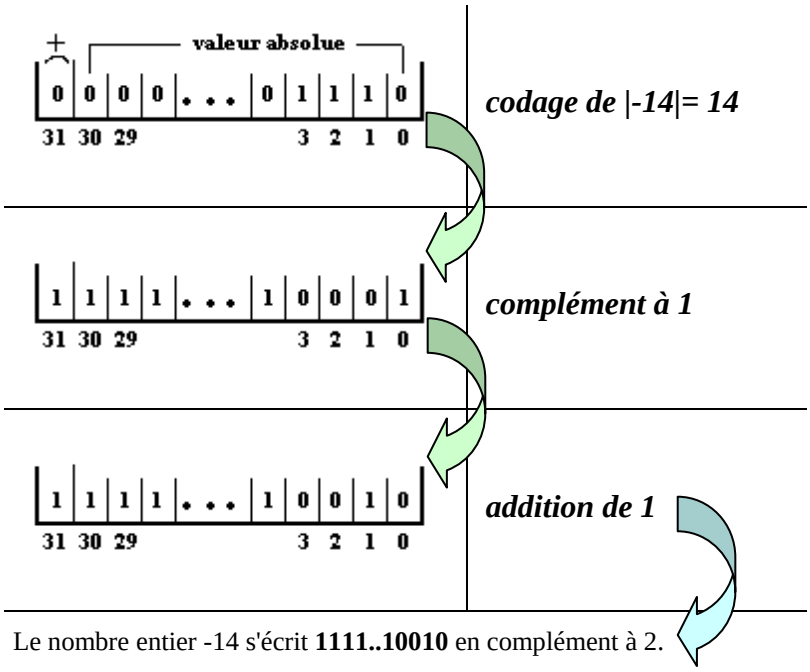
Afin de bien comprendre ces opérateurs, le lecteur doit bien connaître les différents codages des entiers en machine (binaire pur, binaire signé, complément à deux) car les entiers C# sont codés en complément à deux et la manipulation bit à bit nécessite une bonne compréhension de ce codage.

Afin que le lecteur se familiarise bien avec ces opérateurs de bas niveau nous détaillons un exemple pour **chacun d'entre eux**.

Les exemples en 3 instructions C# sur la même mémoire :

Rappel : **int** i = -14 ;

soit à représenter le nombre -14 dans la variable i de type **int** (entier signé sur 32 bits)



Le nombre entier -14 s'écrit **1111..10010** en complément à 2.

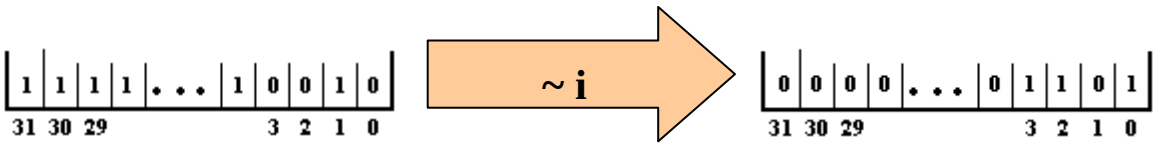
Soient la déclaration C# suivante :

int i = -14 , j ;

Etudions les effets de chaque opérateur bit level sur cette mémoire i.

- Etude de l'instruction : j = ~ i

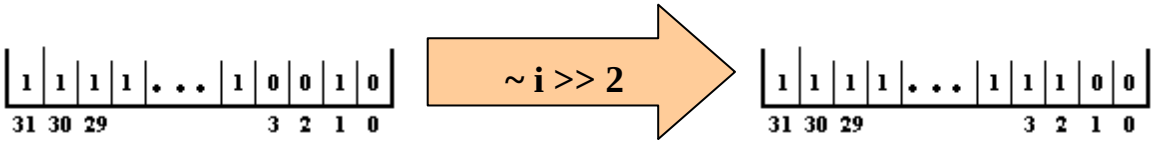
j = ~ i ; *// complémentation des bits de i*



Tous les bits 1 sont transformés en 0 et les bits 0 en 1, puis le résultat est stocké dans j qui contient la valeur 13 (car 000...01101 représente +13 en complément à deux).

- Etude de l'instruction : j = i >> 2

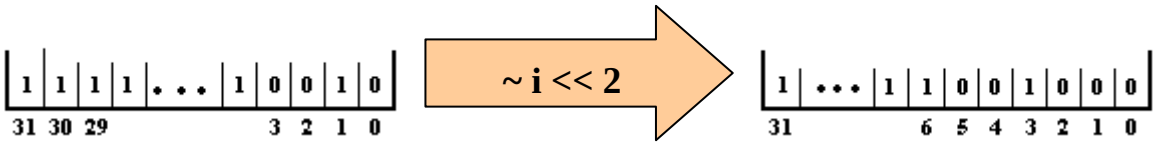
j = i >> 2 ; *// décalage avec signe de 2 bits vers la droite*



Tous les bits sont décalés de 2 positions vers la droite (vers le bit de poids faible), le bit de signe (ici 1) est recopié à partir de la gauche (à partir du bit de poids fort) dans les emplacements libérés (ici le bit 31 et le bit 30), puis le résultat est stocké dans j qui contient la valeur -4 (car 1111...11100 représente -4 en complément à deux).

- Etude de l'instruction : j = i << 2

j = i << 2 ; *// décalage de 2 bits vers la gauche*



Tous les bits sont décalés de 2 positions vers la gauche (vers le bit de poids fort), des 0 sont introduits à partir de la droite (à partir du bit de poids faible) dans les emplacements libérés (ici le bit 0 et le bit 1), puis le résultat est stocké dans j contient la valeur -56 (car 11...1001000 représente -56 en complément à deux).

Exemples opérateurs arithmétiques

```
using System;
namespace CsAlgorithmique
{
class AppliOperat_Arithme
 {
    static void Main(string[ ] args)
    {
      int x = 4, y = 8, z = 3, t = 7, calcul ;
       calcul = x * y - z + t ;
       System.Console.WriteLine(" x * y - z + t = "+calcul);
       calcul = x * y - (z + t) ;
       System.Console.WriteLine(" x * y - (z + t) = "+calcul);
       calcul = x * y % z + t ;
       System.Console.WriteLine(" x * y % z + t = "+calcul);
       calcul = (( x * y) % z ) + t ;
       System.Console.WriteLine("(( x * y) % z ) + t = "+calcul);
       calcul = x * y % ( z + t ) ;
       System.Console.WriteLine(" x * y % ( z + t ) = "+calcul);
       calcul = x *(y % ( z + t ));
       System.Console.WriteLine(" x *( y % ( z + t)) = "+calcul);
    }
 }
}
```

Résultats d'exécution de ce progamme :

```
x * y - z + t = 36
x * y - (z + t) = 22
x * y % z + t = 9
(( x * y) % z ) + t = 9
x * y % ( z + t ) = 2
x *( y % ( z + t)) = 32
```

Exemples opérateurs bit level

```
using System;
namespace CsAlgorithmique
{
class AppliOperat_BitBoole
 {
     static void Main(String[ ] args)
     {
       int x, y, z ,t, calcul=0 ;
       x = 4;  // 00000100
       y = -5; // 11111011
       z = 3;  // 00000011
       t = 7;  // 00000111
       calcul = x & y ;
       System.Console.WriteLine("  x & y  = "+calcul);
       calcul = x & z ;
       System.Console.WriteLine("  x & z  = "+calcul);
       calcul = x & t ;
       System.Console.WriteLine("  x & t  = "+calcul);
       calcul = y & z ;
       System.Console.WriteLine("  y & z  = "+calcul);
       calcul = x | y ;
       System.Console.WriteLine("  x | y  = "+calcul);
       calcul = x | z ;
       System.Console.WriteLine("  x | z  = "+calcul);
       calcul = x | t ;
       System.Console.WriteLine("  x | t  = "+calcul);
       calcul = y | z ;
       System.Console.WriteLine("  y | z  = "+calcul);
       calcul = z ^ t ;
       System.Console.WriteLine("  z ^ t  = "+calcul);
       System.Console.WriteLine(" ~x  = "+~x+",  ~y  = "+~y+",  ~z  = "+~z+",  ~t  = "+~t);
     }
   }
}
```

Résultats d'exécution de ce progamme :

x & y = 0	x | z = 7
x & z = 0	x | t = 7
x & t = 4	y | z = -5
y & z = 3	z ^ t = 4
x | y = -1	~x = -5, ~y = 4, ~z = -4, ~t = -8

Exemples opérateurs bit level - Décalages

```
using System;
namespace CsAlgorithmique
{
class AppliOperat_BitDecalage
 {
  static void Main(String[ ] args)
  {
   int x,y, calcul = 0 ;
   x = -14; // 1...11110010
   y = x;
   calcul = x >>2;   // 1...11111100
   System.Console.WriteLine(" x >> 2  = "+calcul);
   calcul = y <<2 ; // 1...11001000
   System.Console.WriteLine(" y <<2  = "+calcul);
   uint x1,y1, calcul1 = 0 ;
   x1 = 14; // 0...001110
   y1 = x1;
   calcul1 = x1 >> 2;   // 0...000011
   System.Console.WriteLine(" x1 >> 2  = "+calcul1);
   calcul1 = y1 <<2 ; // 0...00111000
   System.Console.WriteLine(" y1 <<2  = "+calcul1);
  }
 }
}
```

Résultats d'exécution de ce progamme :

```
x >> 2  = -4
y <<2  = -56
x1 >> 2  = 3
y1 <<2  = 56
```

Exemples opérateurs booléens

```
using System;
namespace CsAlgorithmique
{
class  AppliOperat_Boole
 {
  static void Main(String[ ] args)
 {
   int x = 4, y = 8, z = 3, t = 7, calcul=0 ;
   bool bool1 ;
   bool1 = x < y;
   System.Console.WriteLine("  x < y  = "+bool1);
   bool1 = (x < y) & (z == t) ;
   System.Console.WriteLine(" (x < y) & (z = = t)  = "+bool1);
   bool1 = (x < y) | (z == t) ;
   System.Console.WriteLine(" (x < y) | (z = = t)  = "+bool1);
   bool1 = (x < y) && (z == t) ;
   System.Console.WriteLine(" (x < y) && (z = = t)  = "+bool1);
   bool1 = (x < y) || (z == t) ;
   System.Console.WriteLine(" (x < y) || (z = = t)  = "+bool1);
   bool1 = (x < y) || ((calcul=z) == t) ;
   System.Console.WriteLine(" (x < y) || ((calcul=z) == t)  = "+bool1+"  ** calcul = "+calcul);
   bool1 = (x < y) | ((calcul=z) == t) ;
   System.Console.WriteLine(" (x < y) | ((calcul=z) == t)  = "+bool1+"  ** calcul = "+calcul);
   System.Console.Read();
  }
 }
}
```

Résultats d'exécution de ce progamme :

```
    x < y  = true
   (x < y) & (z = = t)  = false
   (x < y) | (z = = t)  = true
   (x < y) && (z = = t)  = false
   (x < y) || (z = = t)  = true
   (x < y) || ((calcul=z) == t)  = true  ** calcul = 0
   (x < y) | ((calcul=z) == t)  = true  ** calcul = 3
```

Les instructions

Les instructions de base de C# sont identiques syntaxiquement et sémantiquement à celles de Java, le lecteur qui connaît déjà le fonctionnement des instructions en Java peut ignorer ces chapitres

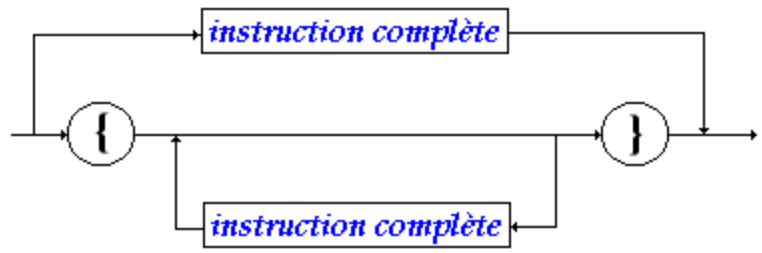
1 - les instructions de bloc

Une large partie de la norme ANSI du langage C est reprise dans C# , ainsi que la norme Delphi.

- Les commentaires sur une ligne débutent par **//....** comme en Delphi
- Les commentaires sur plusieurs lignes sont encadrés par **/* ... */**

Ici, nous expliquons les instructions C# en les comparant à pascal-delphi. Voici la syntaxe d'une instruction en C#:

instruction :



instruction complète :

Toutes les instructions se terminent donc en C# par un point-virgule " ; "

bloc - instruction composée :

L'élément syntaxique

est aussi dénommé **bloc** ou **instruction composée** au sens de la **visibilité** des variables C#.

visibilité dans un bloc - instruction :

Exemple de déclarations licites et de visibilité dans 3 blocs instruction imbriqués :

```
 int a, b = 12;
{ int x , y = 8 ;
  { int z =12;
        x = z ;
        a = x + 1 ;
     { int u = 1 ;
        y = u - b ;
     }
  }
}
```

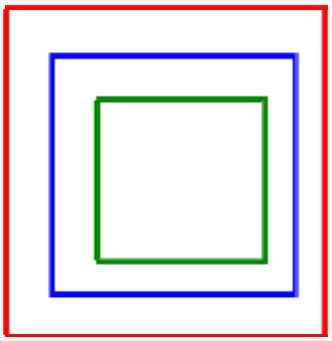


schéma d'imbrication des 3 blocs

Nous examinons ci-dessous l'ensemble des **instructions simples** de C#.

2 - l'affectation

C# est un langage de la famille des langages hybrides, il possède la notion d'instruction d'affectation.

Le symbole d'affectation en C# est " = ", soit par exemple :

```
x = y ;
// x doit obligatoirement être un identificateur de variable.
```

Affectation simple

L'affectation peut être utilisée dans une expression :

soient les instruction suivantes :

```
int a , b = 56 ;
a = (b = 12)+8 ;  // b prend une nouvelle valeur dans l'expression
a = b = c = d =8 ; // affectation multiple
```

simulation d'exécution C# :

instruction	*valeur de a*	*valeur de b*
int a , b = 56 ;	**a = ???**	**b = 56**
a = (b = 12)+8 ;	**a = 20**	**b = 12**

3 - Raccourcis et opérateurs d'affectation

Soit **op** un opérateur appartenant à l'ensemble des opérateurs suivant

{ +, - , * , / , % , << , >> , >>> , & , | , ^ },

Il est possible d'utiliser sur une seule variable le nouvel opérateur **op=** construit avec l'opérateur **op**.

x **op**= y ; signifie en fait : x = x **op** y

Il s'agit plus d'un **raccourci syntaxique** que d'un opérateur nouveau (sa traduction en MSIL est exactement la même : la traduction de a **op**= b devrait être plus courte en instructions p-code que a = a **op** b).

Ci-dessous le code MSIL engendré par i = i+5; et i +=5; est effectivement identique :

Code MSIL engendré	**Instruction C#**
IL_0077: ldloc.1 **IL_0078: ldc.i4.5** **IL_0079: add** **IL_007a: stloc.1**	**i = i + 5 ;**
IL_007b: ldloc.1 **IL_007c: ldc.i4.5** **IL_007d: add** **IL_007e: stloc.1**	**i += 5 ;**

Soient les instruction suivantes :

```
int a , b = 56 ;
a = -8 ;
a += b ;  // équivalent à : a = a + b
b *= 3 ;  // équivalent à : b = b * 3
```

simulation d'exécution C# :

instruction	*valeur de a*	*valeur de b*
int a , b = 56 ;	**a = ???**	**b = 56**
a = -8 ;	**a = -8**	**b = 56**
a += b ;	**a = 48**	**b = 56**
b *= 3 ;	**a = 48**	**b = 168**

Remarques :

- ***Cas d'une optimisation intéressante dans l'instruction suivante :***
  table[f(a)] = table[f(a)] + x ; *// où f(a) est un appel à la fonction f qui serait longue à calculer.*

- ***Si l'on réécrit l'instruction précédente avec l'opérateur += :***
  table[f(a)] += x ; *// l'appel à f(a) n'est effectué qu'une seule fois*

Ci-dessous le code MSIL engendré par "**table[f(i)] = table[f(i)] +9** ;" et "**table[f(i)] += 9** ;" n'est pas le même :

Code MSIL engendré | **Instruction C#**

table[f(i)] = table[f(i)] + 9 ;

```
IL_0086:  ldloc.3 // adr(table)
IL_0087:  ldarg.0
IL_0088:  ldloc.1 // adr(i)
IL_0089:  call      instance int32 exemple.WinForm::f(int32)

IL_008e:  ldloc.3
IL_008f:  ldarg.0
IL_0090:  ldloc.1
IL_0091:  call      instance int32 exemple.WinForm::f(int32)

IL_0096:  ldelem.i4
IL_0097:  ldc.i4.s   9
IL_0099:  add
IL_009a:  stelem.i4
```

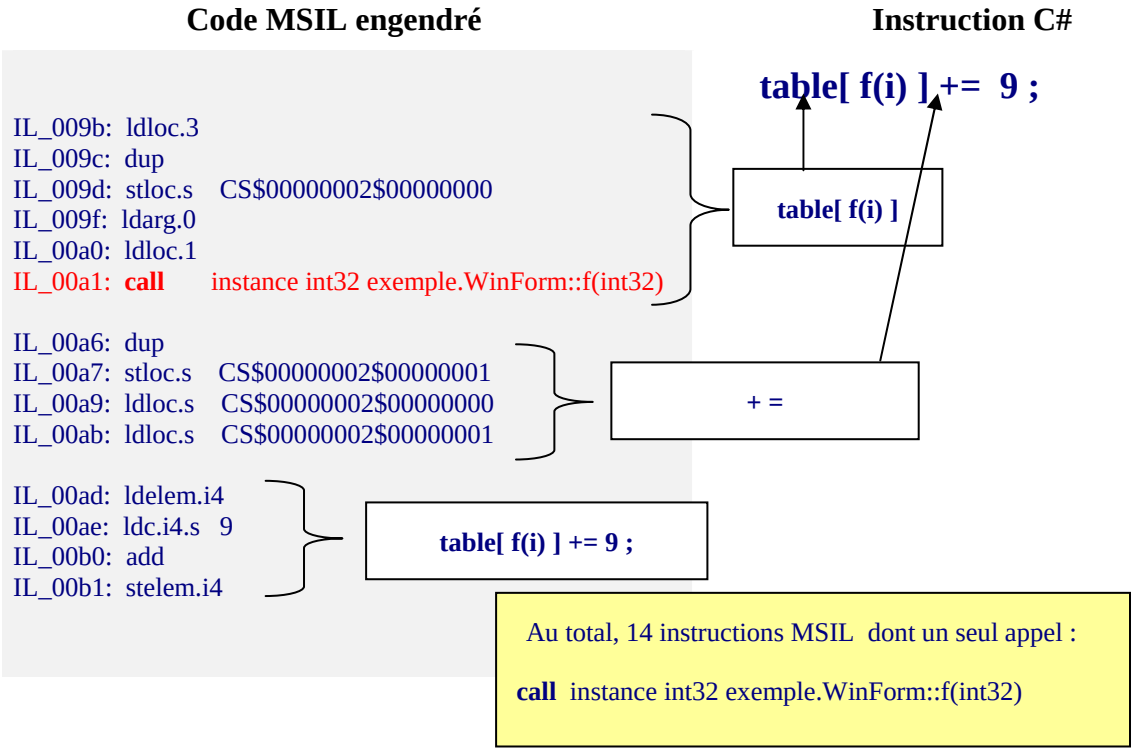
table[f(i)]

table[f(i)]

table[f(i)] = table[f(i)] + 9 ; (suite)

Au total, 12 instructions MSIL dont deux appels :

call instance int32 exemple.WinForm::f(int32)



Dans l'exemple qui précède, il y a réellement gain sur le temps d'exécution de l'instruction **table[f(i)] += 9**, si le temps d'exécution de l'appel à f(i) à travers l'instruction MSIL < **call** instance int32 exemple.WinForm::f(int32) > , est significativement long devant les temps d'exécution des opérations **ldloc** et **stloc**.

En fait d'une manière générale en C# comme dans les autres langages, il est préférable d'adopter l'attitude prise en Delphi qui consiste à encourager la lisibilité du code en ne cherchant pas à écrire du code le plus court possible. Dans notre exemple précédent, la simplicité consisterait à utiliser une variable locale **x** et à stocker la valeur de f(i) dans cette variable :

table[f(i)] = table[f(i)] + 9 ;	**x = f(i) ;** **table[x] = table[x] + 9 ;**

Ces deux écritures étant équivalentes seulement si f(i) ne contient aucun effet de bord !

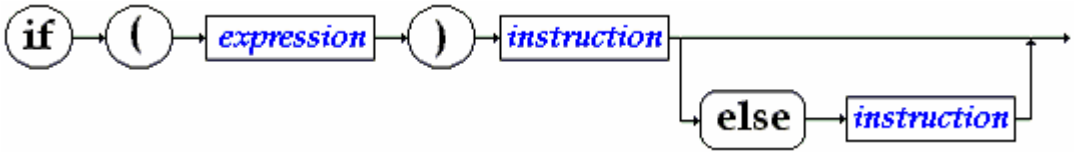
Info MSIL :

> **ldloc** : Charge la variable locale à un index spécifique dans la pile d'évaluation.
>
> **stloc** : Dépile la pile d'évaluation et la stocke dans la liste de variables locales à un index spécifié.

Les instructions conditionnelles

1 - l'instruction conditionnelle

Syntaxe :



Schématiquement les conditions sont de deux sortes :

- **if** (Expr) Instr ;
- **if** (Expr) Instr ; **else** Instr ;

La définition de l'instruction conditionnelle de C# est classiquement celle des langages algorithmiques; comme en pascal l'expression doit être de type booléen (différent du C), la notion d'instruction a été définie plus haut.

Exemple d'utilisation du if..else (comparaison avec Delphi)

Pascal-Delphi	C#

var a , b , c : integer ;
....
if b=0 **then** c := 1
else begin
c := a / b;
writeln("c = ",c);
end;
c := a*b ;
if c <>0 **then** c:= c+b
else c := a
	int a , b , c ;
	....
	if (b = = 0) c =1 ;
	else {
	c = a / b;
	System.Console.WriteLine ("c = " + c);
	}
	if ((c = a*b) != 0) c += b;
	else c = a;

Remarques :

- L'instruction " **if** ((c = a*b) != 0) c +=b; **else** c = a; " contient une affectation intégrée dans le test afin de vous montrer les possibilités de C# : la valeur de a*b est rangée dans c avant d'effectuer le test sur c.

- Comme Delphi, C# contient le manque de fermeture des instructions conditionnelles ce qui engendre le classique problème du dandling **else** d'algol, c'est le compilateur qui résout l'ambiguïté par rattachement du **else** au dernier **if** rencontré (évaluation par la gauche.

  L'instruction suivante est ambiguë :

  if (Expr1) **if** (Expr2) InstrA ; **else** InstrB ;

  Le compilateur résoud l'ambigüité de cette instruction ainsi (rattachement du **else** au dernier **if**):

  if (Expr1) **if** (Expr2) InstrA ; **else** InstrB ;

- Comme en pascal, si l'on veut que l'instruction **else** InstrB ; soit rattachée au premier **if**, il est nécessaire de parenthéser (introduire un bloc) le second **if** :

Exemple de parenthésage du else pendant

Pascal-Delphi	C#

if Expr1 **then**
begin
if Expr2 **then** InstrA
end
else InstrB
	if (Expr1) **{**
	if (Expr2) InstrA ;
	}
	else InstrB

2 - l'opérateur conditionnel

Il s'agit ici comme dans le cas des opérateurs d'affectation d'une sorte de raccourci entre l'opérateur conditionnel **if**...**else** et l'affectation. Le but étant encore d'optimiser le MSIL engendré.

Syntaxe :

Où ***expression*** renvoie une valeur booléenne (le test), les deux termes ***valeur*** sont des expressions générales (variable, expression numérique, boolénne etc...) renvoyant une valeur de type quelconque.

Sémantique :

Exemple :

```
int a,b,c ;
c =  a = = 0  ?  b  :  a+1 ;
```

Si l'***expression*** est **true** l'opérateur renvoie la première valeur, (dans l'exemple c vaut la valeur de b)
Si l'***expression*** est **false** l'opérateur renvoie la seconde valeur (dans l'exemple c vaut la valeur de a+1).

Sémantique de l'exemple avec un **if**..**else** :

```
if (a = = 0) c = b; else c = a+1;
```

Code MSIL engendré	Instruction C#
IL_0007: ldloc.0 IL_0008: brfalse.s IL_000f IL_000a: ldloc.0 IL_000b: ldc.i4.1 IL_000c: add IL_000d: br.s IL_0010 IL_000f: ldloc.1 IL_0010: **stloc.2**	Opérateur conditionnel : **c = a == 0 ? b : a+1 ;** une seule opération de stockage pour c : IL_0010: **stloc.2**
IL_0011: ldloc.0 IL_0012: brtrue.s IL_0018 IL_0014: ldloc.1 IL_0015: **stloc.2** IL_0016: br.s IL_001c IL_0018: ldloc.0 IL_0019: ldc.i4.1 IL_001a: add IL_001b: **stloc.2**	Instruction conditionnelle : **if** (a = = 0) c = b; **else** c = a+1; deux opérations de stockage pour c : IL_0015: **stloc.2** IL_001b: **stloc.2**

Le code MSIL engendré a la même structure classique de code de test pour les deux instructions, la traduction de l'opérateur sera légèrement plus rapide que celle de l'instructions car, il n'y a pas besoin de stocker deux fois le résultat du test dans la variable c (qui ici, est représentée par l'instruction MSIL **stloc.2**)

Question : utiliser l'opérateur conditionnel pour calculer le plus grand de deux entiers.

> ***réponse :***
> ***int*** *a , b , c ; ...*
> *c = a>b ? a : b ;*

Question : que fait ce morceau le programme ci-après ?

int a , b , c ; ....
c = a>b ? (b=a) : (a=b) ;

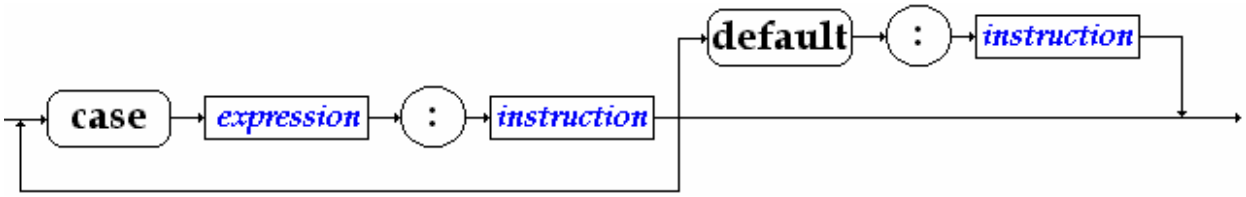
> ***réponse :***
> *a,b,c contiennent après exécution le plus grand des deux entiers contenus au départ dans a et b.*

3 - l'opérateur switch...case

Syntaxe :

switch :

bloc switch :



Sémantique :

- La partie expression d'une instruction switch doit être une expression ou une variable du type **byte**, **char**, **int**, **short**, **string** ou bien **enum**.
- La partie expression d'un bloc switch doit être une constante ou une valeur immédiate du type **byte**, **char**, **int**, **short**, **string** ou bien **enum**.
- **switch** <Epr1> s'appelle la partie sélection de l'instruction : il y a évaluation de <Epr1> puis selon la valeur obtenue le programme s'exécute en séquence à partir du case contenant la valeur immédiate égale. Il s'agit donc d'un déroutement du programme, dès que <Epr1> est évaluée, vers l'instruction étiquetée par le case <Epr1> associé.

*Cette instruction en C#, contrairement à Java, est structurée , elle doit **obligatoirement** être utilisée avec l'instruction **break** afin de simuler le comportement de l'instruction structurée **case..of** du pascal.*

Exemple de switch..case..break

Pascal-Delphi	C#
var x : **char** ; .... **case** x of 'a' : InstrA; 'b' : InstrB; **else** InstrElse **end**;	**char** x ; .... **switch** (x) **{** **case** 'a' : InstrA ; **break**; **case** 'b' : InstrB ; **break**; **default** : InstrElse; **break**; **}**

Dans ce cas le déroulement de l'instruction **switch** après déroutement vers le bon **case**, est interrompu par le **break** qui renvoie la suite de l'exécution après la fin du **bloc switch**. Une telle utilisation correspond à une utilisation de if...else imbriqués (donc une utilisation structurée) mais devient plus lisible que les **if ..else** imbriqués, elle est donc fortement conseillée dans ce cas.

Exemples :

C# - source	Résultats de l'exécution
int x = 10; **switch** (x+1) { **case** 11 : System.Console.WriteLine (">> case 11"); **break**; **case** 12 : System.Console.WriteLine (">> case 12"); **break**; **default** : System.Console.WriteLine (">> default")"); **break**; }	>> case 11
int x = 11; **switch** (x+1) { **case** 11 : System.Console.WriteLine (">> case 11"); **break**; **case** 12 : System.Console.WriteLine (">> case 12"); **break**; **default** : System.Console.WriteLine (">> default")"); **break**; }	>> case 12

Il est toujours possible d'utiliser des instructions **if** … **else** imbriquées pour représenter un **switch** avec **break** :

Programmes équivalents switch et if...else :

C# - switch	C# - if...else
int x = 10; **switch** (x+1) { **case** 11 : System.Console.WriteLine (">> case 11"); **break**; **case** 12 : System.Console.WriteLine (">> case 12"); **break**; **default** : System.Console.WriteLine (">> default"); **break**; }	**int** x = 10; **if** (x+1= = 11) System.Console.WriteLine (">> case 11"); **else** **if** (x+1= = 12) System.Console.WriteLine (">> case 12"); **else** System.Console.WriteLine (">> default");

Bien que la syntaxe du **switch …break** soit plus contraignante que celle du **case…of** de Delphi, le fait que cette instruction apporte commme le **case…of** une structuration du code, conduit à une amélioration du code et augmente sa lisibilité. Lorsque cela est possible, il est donc conseillé de l'utiliser d'une manière générale comme alternative à des **if...then…else** imbriqués.

Les instructions itératives

1 - l'instruction while

Syntaxe :

Où expression est une ***expression*** renvoyant une valeur booléenne (le test de l'itération).

Sémantique :
Identique à celle du pascal (instruction algorithmique **tantque** .. **faire** .. **ftant**) avec le même défaut de fermeture de la boucle.

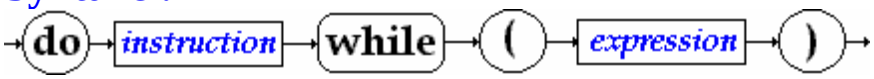
Exemple de boucle while

Pascal-Delphi	C#
while Expr **do** Instr	**while (** Expr **)** Instr ;

while Expr **do**
begin
InstrA ;
InstrB ; ...
end
	while (Expr **)**
	{
	InstrA ;
	InstrB ; ...
	}

2 - l'instruction do ... while

Syntaxe :



Où expression est une ***expression*** renvoyant une valeur booléenne (le test de l'itération).

Sémantique :
L'instruction "**do** Instr **while** (Expr)" fonctionne comme l'instruction algorithmique **<u>répéter</u>** Instr **<u>jusquà</u> non** Expr.

Sa sémantique peut aussi être expliquée à l'aide d'une autre instruction C#, le **while( )** :

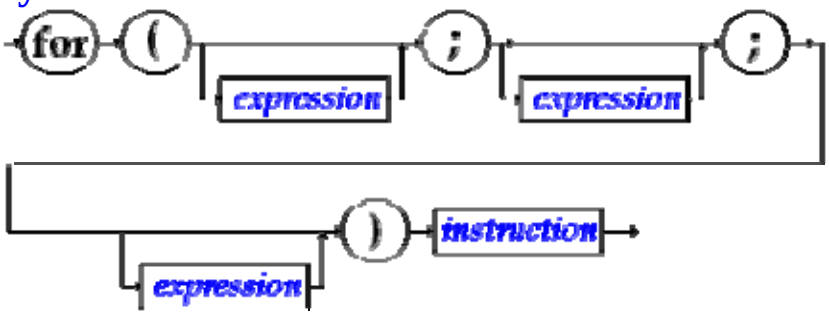
do Instr **while** (Expr) <=> Instr ; **while** (Expr) Instr

Exemple de boucle do...while

Pascal-Delphi	C#
repeat InstrA ; InstrB ; ... **until not** Expr	**do** **{** InstrA ; InstrB ; ... **} while (** Expr **)**

3 - l'instruction for(...)

Syntaxe :



Sémantique :
Une boucle **for** contient 3 expressions **for** (**Expr1** ; **Expr2** ; **Expr3**) Instr, d'une manière générale chacune de ces expressions joue un rôle différent dans l'instruction **for.** Une instruction for en C# (comme en C) est plus puissante et plus riche qu'une boucle **for** dans d'autres langages algorithmiques. Nous donnons ci-après une sémantique minimale :

- **Expr1** sert à initialiser une ou plusieurs variables (dont éventuellement la variable de contrôle de la boucle) sous forme d'une liste d'instructions d'initialisation séparées par des virgules.
- **Expr2** sert à donner la condition de rebouclage sous la fome d'une expression renvoyant une valeur booléenne (le test de l'itération).
- **Expr3** sert à réactualiser les variables (dont éventuellement la variable de contrôle de la boucle)sous forme d'une liste d'instructions séparées par des virgules.

L'instruction "**for** (**Expr1** ; **Expr2** ; **Expr3**) Instr" fonctionne au minimum comme l'instruction algorithmique **pour**... **fpour**, elle est toutefois plus puissante que cette dernière.

Sa sémantique peut aussi être approximativement**(*)** expliquée à l'aide d'une autre instruction C# **while** :

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ; **while** (**Expr2**) { Instr ; **Expr3** }

(*)Nous verrons au paragraphe consacré à l'instruction **continue** que si l'instruction **for** contient un **continue** cette définition sémantique n'est pas valide.

Exemples montrant la puissance du for

Pascal-Delphi	C#

for i:=1 **to** 10 **do**
begin
InstrA ;
InstrB ; ...
end
	for (**i = 1**; **i<=10**; **i++**)
	{
	InstrA ;
	InstrB ; ...
	}

i := 10; k := i;
while (i>-450) **do**
begin
InstrA ;
InstrB ; ...
k := k+i;
i := i-15;
end
	for (**i = 10, k = i** ;**i>-450** ; **k += i , i -= 15**)
	{
	InstrA ;
	InstrB ; ...
	}

i := n;
while i<>1 **do**
if i **mod** 2 = 0 **then** i := i **div** 2
else i := i+1
	int i, j ;
	
	for (**i = n, j** ;**i !=1** ; **j = i % 2 == 0 ? i /=2 : i++**);
	// pas de corps de boucle !

- Le premier exemple montre une boucle for classique avec la variable de contrôle "i" (indice de boucle), sa borne initiale "i=1" et sa borne finale "10", le pas d'incrémentation séquentiel étant de 1.
- Le second exemple montre une boucle toujours contrôlée par une variable "i", mais dont le pas de décrémentation séquentiel est de -15.
- Le troisème exemple montre une boucle aussi contrôlée par une variable "i", mais dont la variation n'est pas séquentielle puisque la valeur de i est modifiée selon sa parité (**i % 2 == 0 ? i /=2 : i++**).

Voici un exemple de boucle **for** dite **boucle infinie** :

```
for ( ; ; );   est équivalente à  while (true);
```

Voici une boucle ne possédant pas de variable de contrôle(f(x) est une fonction déjà déclarée) :

```
for (int n=0 ; Math.abs(x-y) < eps ; x = f(x) );
```

Terminons par une boucle for possédant deux variables de contrôle :

```
//inverse d'une suite de caractère dans un tableau par permutation des deux
extrêmes
char [ ] Tablecar ={'a','b','c','d','e','f'} ;
for ( i = 0 , j = 5 ;  i<j ; i++ , j-- )
{ char car ;
   car = Tablecar[i];
   Tablecar[i ]= Tablecar[j];
   Tablecar[j] = car;
 }
```

dans cette dernière boucle ce sont les variations de i et de j qui contrôlent la boucle.

Remarques récapitulatives sur la boucle for en C# :

rien n'oblige à incrémenter ou décrémenter la variable de contrôle,

rien n'oblige à avoir une instruction à exécuter (corps de boucle),

rien n'oblige à avoir une variable de contrôle,

rien n'oblige à n'avoir qu'une seule variable de contrôle.

Les instructions de rupture de séquence

1 - l'instruction d'interruption break

Syntaxe :

Sémantique :
Une instruction **break** ne peut se situer qu'à l'intérieur du corps d'instruction d'un bloc **switch** ou de l'une des trois itérations **while**, **do..while**, **for**.

Lorsque **break** est présente dans l'une des trois itérations **while**, **do..while**, **for** :

> **break** interrompt l'exécution de la boucle dans laquelle elle se trouve, l'exécution se poursuit après le corps d'instruction.

Exemple d'utilisation du break dans un for :
(recherche séquentielle dans un tableau)

```
int [ ] table = {12,-5,7,8,-6,6,4,78};
int elt = 4;
for ( i = 0 ; i<8 ; i++ )
  if (elt= =table[i]) break ;
if (i = = 8)System.Console.WriteLine("valeur : "+elt+" pas trouvée.");
else System. Console.WriteLine("valeur : "+elt+" trouvée au rang :"+i);
```

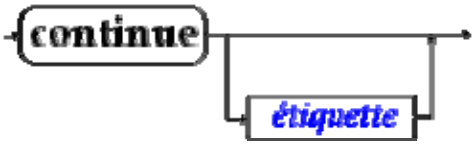
Explications

> Si la valeur de la variable elt est présente dans le tableau table, l'expression (elt= =table[i]) est true et **break** est exécutée (arrêt de la boucle et exécution de **if** (i = = 8)...).

Après l'exécution de la boucle **for**, lorsque l'instruction **if** (i = = 8)... est exécutée, soit la boucle s'est exécutée complètement (recherche infructueuse), soit le **break** l'a arrêtée prématurément (elt est trouvé dans le tableau).

2 - l'instruction de rebouclage continue

Syntaxe :



Sémantique :
Une instruction **continue** ne peut se situer qu'à l'intérieur du corps d'instruction de l'une des trois itérations **while**, **do..while**, **for**.

Lorsque **continue** est présente dans l'une des trois itérations **while**, **do..while**, **for** :

- Si **continue** n'est pas suivi d'une étiquette elle interrompt l'exécution de la séquence des instructions situées après elle, l'exécution se poursuit par rebouclage de la boucle. Elle agit comme si l'on venait d'exécuter la dernière instruction du corps de la boucle.
- Si **continue** est suivi d'une étiquette elle fonctionne comme un **goto** (utilisation **déconseillée** en programmation moderne, c'est pourquoi nous n'en dirons pas plus !)

Exemple d'utilisation du continue dans un for :

```
int [ ] ta = {12,-5,7,8,-6,6,4,78}, tb = new int[8];
for ( i = 0, n = 0 ; i<8 ;  i++ , k = 2*n )
{ if ( ta[i] = = 0 ) continue ;
  tb[n] = ta[i];
  n++;
}
```

Explications

Rappelons qu'un **for** s'écrit généralement :
for (**Expr1** ; **Expr2** ; **Expr3**) Instr
L'instruction **continue** présente dans une telle boucle **for** s'effectue ainsi :

- exécution immédiate de **Expr3**
- ensuite, exécution de **Expr2**

- réexécution du corps de boucle.

Si l'expression (ta[i] = = 0) est true, la suite du corps des instructions de la boucle (tb[n] = ta[i]; n++;) n'est pas exécutée et il y a rebouclage du **for** .

Le déroulement est alors le suivant :

- **i++ , k = 2*n** en premier ,
- puis la condition de rebouclage : **i<8**

et la boucle se poursuit en fonction de la valeur de la condition de rebouclage.

Cette boucle recopie dans le tableau d'entiers **tb** les valeurs non nulles du tableau d'entiers **ta**.

Attention

Nous avons déjà signalé plus haut que l'équivalence suivante entre un **for** et un **while**

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ; **while** (**Expr2**) { Instr ; **Expr3** }

valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue**.

Voyons ce qu'il en est en reprenant l'exemple précédent. Essayons d'écrire la boucle **while** qui lui serait équivalente selon la définition générale. Voici ce que l'on obtiendrait :

for (**i = 0, n = 0** ; **i<8** ; **i++ , k = 2*n**)
{ **if** (ta[i] = = 0) **continue** ;
tb[n] = ta[i];
n++;
}
	i = 0; n = 0 ;
	while (**i<8**)
	{ **if** (ta[i] = = 0) **continue** ;
	tb[n] = ta[i];
	n++;
	i++ ; k = 2*n;
	}

Dans le **while** le **continue** réexécute la condition de rebouclage **i<8** sans exécuter l'expression **i++ ; k = 2*n;** (nous avons d'ailleurs ici une boucle infinie).

Une boucle **while** strictement équivalente au **for** précédent pourrait être la suivante :

for (**i = 0, n = 0** ; **i<8** ; **i++ , k = 2*n**)
{ **if** (ta[i] = = 0) **continue** ;
tb[n] = ta[i];
n++;
}
	i = 0; n = 0 ;
	while (**i<8**)
	{ **if** (ta[i] = = 0)
	{ **i++ ; k = 2*n;**
	continue ;
	}
	tb[n] = ta[i];
	n++;
	i++ ; k = 2*n;
	}

Classes avec méthodes static

Une classe suffit
Les méthodes sont des fonctions
Transmission des paramètres en C# ***(plus riche qu'en java)***
Visibilité des variables

Avant d'utiliser les possibilités offertes par les classes et les objets en C#, apprenons à utiliser et exécuter des applications simples C# ne nécessitant pas la construction de nouveaux objets, ni de navigateur pour s'exécuter

Comme C# est un langage entièrement orienté objet, un programme C# est composé de plusieurs classes, nous nous limiterons à une seule classe.

1 - Une classe suffit

On peut très grossièrement assimiler un programme C# ne possédant qu'une seule classe, à un programme principal classique d'un langage de programmation algorithmique.

- Une classe minimale commence obligatoirement par le mot **class** suivi de l'identificateur de la classe puis du corps d'implémentation de la classe dénommé **bloc de classe**.
- Le bloc de classe est parenthésé par deux accolades "**{**" et "**}**".

Syntaxe d'une classe exécutable

Exemple1 de classe minimale :

```
class Exemple1 {  }
```

Cette classe ne fait rien et ne produit rien.

En fait, une classe quelconque peut s'exécuter toute seule à condition qu'elle possède dans ses déclarations internes la méthode **Main** (avec ou sans paramètres) qui sert à lancer l'exécution de la classe (fonctionnement semblable au lancement d'un programme principal).

Exemple2 de squelette d'une classe minimale exécutable :

```
class Exemple2
{
   static void Main( ) // sans paramètres
  { // c'est ici que vous écrivez votre programme principal
   }
}
```

Exemple3 trivial d'une classe minimale exécutable :

```
class Exemple3
{
   static void Main(string[ ] args) // avec paramètres
  { System.Console.WriteLine("Bonjour !");
   }
}
```

Exemples d'applications à une seule classe

Nous reprenons deux exemples de programme utilisant la boucle **for**, déjà donnés au chapitre sur les instructions, cette fois-ci nous les réécrirons sous la forme d'une application exécutable.

Exemple1

```
class Application1
{
static void Main(string[ ] args)
  { /* inverse d'une suite de caractère dans un tableau par
      permutation des deux extrêmes */
   char [ ] Tablecar ={'a','b','c','d','e','f'} ;
   int i, j ;
   System.Console.WriteLine("tableau avant : " + new string(Tablecar));
   for ( i = 0 , j = 5 ;  i<j ; i++ , j-- )
    { char car ;
      car = Tablecar[i];
      Tablecar[i ]= Tablecar[j];
      Tablecar[j] = car;
    }
   System.Console.WriteLine("tableau après : " + new string(Tablecar));
  }
}
```

L'instruction "**new** string(Tablecar)" sert uniquement pour l'affichage, elle crée une string à partir du tableau de **char**.

Contrairement à java il n'est pas nécessaire en C#, de sauvegarder la classe dans un fichier qui porte le même nom, tout nom de fichier est accepté à condition que le suffixe soit **cs,** ici "**AppliExo1**.**cs**". Lorsque l'on demande la compilation (production du bytecode) de ce fichier source "**AppliExo1**.**cs**" le fichier cible produit en **bytecode MSIL** se dénomme "**AppliExo1**.**exe**", il est alors prêt à être exécuté par la **machine virtuelle du CLR**.

Le résultat de l'exécution de ce programme est le suivant :

> *tableau avant* : abcdef
> *tableau après* : fedcba

Exemple2

```
class Application2
{
  static void Main(string[ ] args)
  { // recherche séquentielle dans un tableau
    int [ ] table= {12,-5,7,8,-6,6,4,78};
    int elt = 4, i ;
    for ( i = 0 ; i<8 ; i++ )
      if (elt= =table[i]) break ;
    if (i = = 8) System.Console.WriteLine("valeur : "+elt+" pas trouvée.");
    else System.Console.WriteLine ("valeur : "+elt+" trouvée au rang :"+i);
  }
}
```

Après avoir sauvegardé la classe dans un fichier xxx.cs, ici dans notre exemple "**AppliExo2**.**cs**", la compilation de ce fichier "**AppliExo2**.**cs**" produit le fichier "**AppliExo2**.**exe**" prêt à être exécuté par la **machine virtuelle du CLR**.

Le résultat de l'exécution de ce programme est le suivant :

```
valeur : 4 trouvée au rang : 6
```

Conseil de travail :

> Reprenez tous les exemples simples du chapitre sur les instructions de boucle et le switch en les intégrant dans une seule classe (comme nous venons de le faire avec les deux exemples précédents) et exécutez votre programme.

2 - Les méthodes sont des fonctions

Les méthodes ou fonctions représentent une encapsulation des instructions qui déterminent le fonctionnement d'une classe. Sans méthodes pour agir, une classe ne fait rien de particulier, dans ce cas elle ne fait que contenir des attributs.

Méthode élémentaire de classe

Bien que C# distingue deux sortes de méthodes : les **méthodes de classe** et les **méthodes d'instance**, pour l'instant dans cette première partie nous décidons à titre pédagogique et simplificateur de n'utiliser que **les méthodes de classe**, le chapitre sur C# et la programmation orientée objet apportera les compléments adéquats.

Une méthode de classe commence **obligatoirement** par le mot clef **static**.

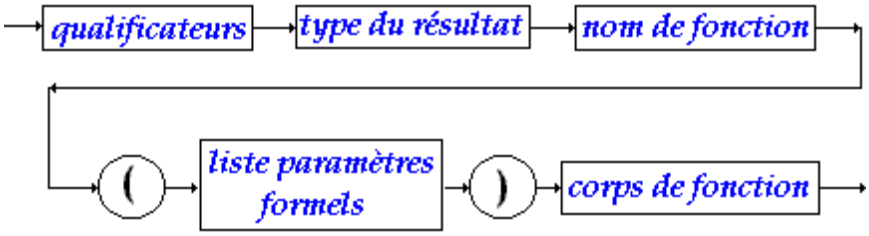
Donc par la suite dans ce document lorsque nous emploierons le mot méthode sans autre adjectif, il s'agira d'une **méthode de classe**, comme nos applications ne possèdent qu'une seule classe, nous pouvons assimiler ces méthodes aux fonctions de l'application et ainsi retrouver une *utilisation classique de C# en mode application*.

> **Attention**, il est impossible en C# de déclarer une méthode à l'intérieur d'une autre méthode comme en pascal; toutes les méthodes sont au même niveau de déclaration : ce sont les méthodes de la classe !

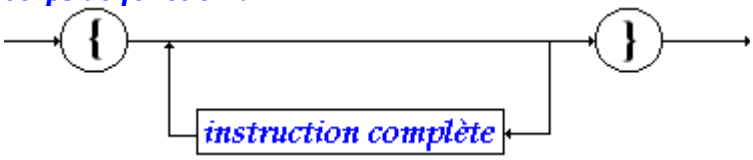
Déclaration d'une méthode

La notion de fonction en C# est semblable à celle de Java, elle comporte **une en-tête** avec des paramètres formels **et un corps de fonction** ou de méthode qui contient les instructions de la méthode qui seront exécutés lors de son appel. La déclaration et l'implémentation doivent être consécutives comme l'indique la syntaxe ci-dessous :

Syntaxe :



corps de fonction :



Nous dénommons en-tête de fonction la partie suivante :
<qualificateurs><type du résultat><nom de fonction> **(**<liste paramètres formels>**)**

Sémantique :

- Les qualificateurs sont des mots clefs permettant de modifier la ***visibilité*** ou le ***fonctionnement*** d'une méthode, nous n'en utiliserons pour l'instant qu'un seul : le mot clef **static** permettant de désigner la méthode qu'il qualifie comme une méthode de classe dans la classe où elle est déclarée. Une méthode n'est pas nécessairement qualifiée donc ce mot clef peut être omis.
- Une méthode peut renvoyer un résultat d'un type C# quelconque en particulier d'un des types élémentaires (**int**, **byte**, **short**, **long**, **bool**, **double**, **float**, **char...**) et nous verrons plus loin qu'elle peut renvoyer un résultat de type objet comme en Delphi. Ce mot clef ne doit pas être omis.
- Il existe en C# comme en C une écriture fonctionnelle correspondant aux procédures des langages procéduraux : on utilise une **fonction qui ne renvoie aucun résultat**. L'approche est inverse à celle du pascal où la procédure est le bloc fonctionnel de base et la fonction n'en est qu'un cas particulier. En C# la fonction (ou méthode) est le seul bloc fonctionnel de base et la procédure n'est qu'un cas particulier de fonction dont le retour est de type **void**.
- La liste des paramètres formels est semblable à la partie déclaration de variables en C# (sans initialisation automatique). La liste peut être vide.
- Le corps de fonction est identique au bloc instruction C# déjà défini auparavant. Le corps de fonction peut être vide (la méthode ne représente alors aucun intérêt).

Exemples d'en-tête de méthodes ***sans paramètres*** en C#

int calculer( ){.....}	renvoie un entier de type **int**
bool tester( ){.....}	renvoie un entier de type **bool**
void uncalcul( ){.....}	procédure **ne renvoyant rien**

Exemples d'en-tête de méthodes ***avec paramètres*** en C#

int calculer(**byte** a, **byte** b, **int** x) {.....}	**fonction à 3 paramètres**
bool tester(**int** k) {.....}	**fonction à 1 paramètre**
void uncalcul(**int** x, **int** y, **int** z) {.....}	**procédure à 3 paramètres**

Appel d'une méthode

L'**appel de méthode** en C# s'effectue très classiquement avec des paramètres effectifs dont le **nombre** doit obligatoirement être le **même** que celui des paramètres formels et le **type** doit être soit le **même**, soit un type **compatible** ne nécessitant pas de transtypage.

Exemple d'appel de méthode-procédure ***sans paramètres*** en C#

```
class Application3
{
   static void Main(string[ ] args)
  {
            afficher( );
  }
  static void afficher( )
 {
   System.Console.WriteLine("Bonjour");
  }
}
```

Appel de la méthode afficher

Exemple d'appel de méthode-procédure ***avec paramètres de même type*** en C#

```
class Application4
{  static void Main(string[ ] args)
    { // recherche séquentielle dans un tableau
      int [ ] table= {12,-5,7,8,-6,6,4,78};
      long elt = 4;
      int i ;
      for ( i = 0 ; i<=8 ; i++ )
      if (elt == table[i]) break ;
      afficher(i,elt);
    }
  static void afficher (int rang , long val)
  { if (rang == 8)
      System.Console.WriteLine ("valeur : "+ val+" pas trouvée.");
    else
     System.Console.WriteLine ("valeur : "+ val
                +" trouvée au rang :"+ rang);
  }
}
```

Appel de la méthode afficher

Afficher (**i** , **elt**);

Les deux paramètres effectifs "**i**" et "**elt**" sont du même type que le paramètre formel associé.

- Le paramètre effectif "**i**" est associé au paramètre formel **rang**.

- Le paramètre effectif "**elt**" est associé au paramètre formel **val**.

3 - Transmission des paramètres

Rappelons tout d'abord quelques principes de base :

Dans tous les langages possédant la notion de sous-programme (ou fonction ou procédure), il se pose une question à savoir : à quoi servent les paramètres formels ? Les paramètres **formels** décrits lors de la déclaration d'un sous-programme ne sont que des variables muettes servant à expliquer le fonctionnement du sous-programme sur des futures variables lorsque le sous-programme s'exécutera **effectivement**.

La démarche en informatique est semblable à celle qui, en mathématiques, consiste à écrire la fonction f(x) = 3*x - 7, dans laquelle x est une variable muette indiquant comment f est calculée : en informatique **elle joue le rôle du paramètre formel**. Lorsque l'on veut obtenir une valeur effective de la fonction mathématique f, par exemple pour x=2, on écrit f(2) et l'on calcule f(2)=3*2 - 7 = -1. En informatique on "**passera**" un paramètre effectif dont la valeur vaut 2 à la fonction. D'une manière générale, en informatique, il y a un sous-programme appelant et un sous-programme appelé par le sous-programme appelant.

Compatibilité des types des paramètres

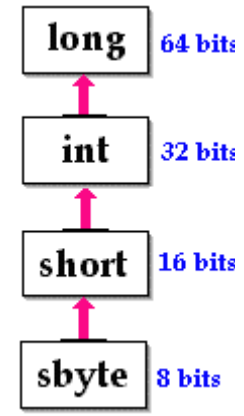
Resituons la compatibilité des types entier et réel en C#.

Un moyen mnémotechnique pour retenir cette compatibilité est indiqué dans les figures ci-dessous, par la taille en nombre décroissant de bits de chaque type que l'on peut mémoriser sous la forme "**qui peut le plus peut le moins**" ou bien un type à **n** bits accueillera un sous-type à **p** bits, si **p** est inférieur à **n**.

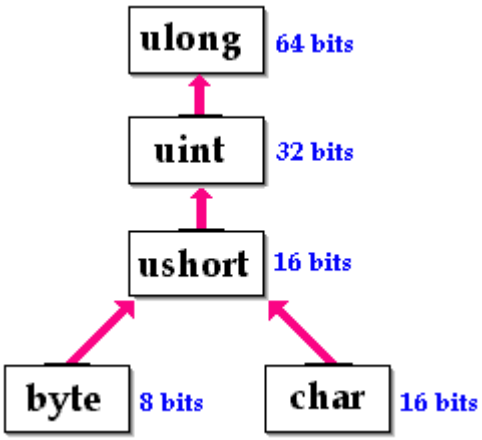
Compatibilité ascendante des types de variables en C#

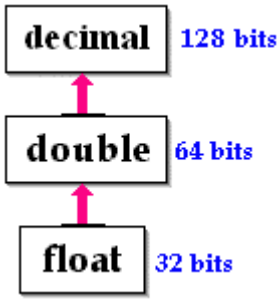
(conversion implicite sans transtypage obligatoire)

Les types entiers signés :



Les types entiers non signés :





Exemple d'appel de la même méthode-procédure ***avec paramètres de type compatibles*** en C#

```
class Application5
{  static void Main(string[ ] args)
    { // recherche séquentielle dans un tableau
      int [ ] table= {12,-5,7,8,-6,6,4,78};
      sbyte elt = 4;
      short i ;
      for ( i = 0 ; i<8 ; i++ )
      if (elt= =table[i]) break ;
      afficher(i,elt);
    }
  static void afficher (int rang , long val)
   { if (rang == 8)
       System.Console.WriteLine ("valeur : "+ val+" pas trouvée.");
     else
      System.Console.WriteLine ("valeur : "+ val
                                          +" trouvée au rang :"+ rang);
  }
}
```

Appel de la méthode afficher

afficher(**i**,**elt**);

Les deux paramètres effectifs "**i**" et "**elt**" sont d'un type compatible avec celui du paramètre formel associé.

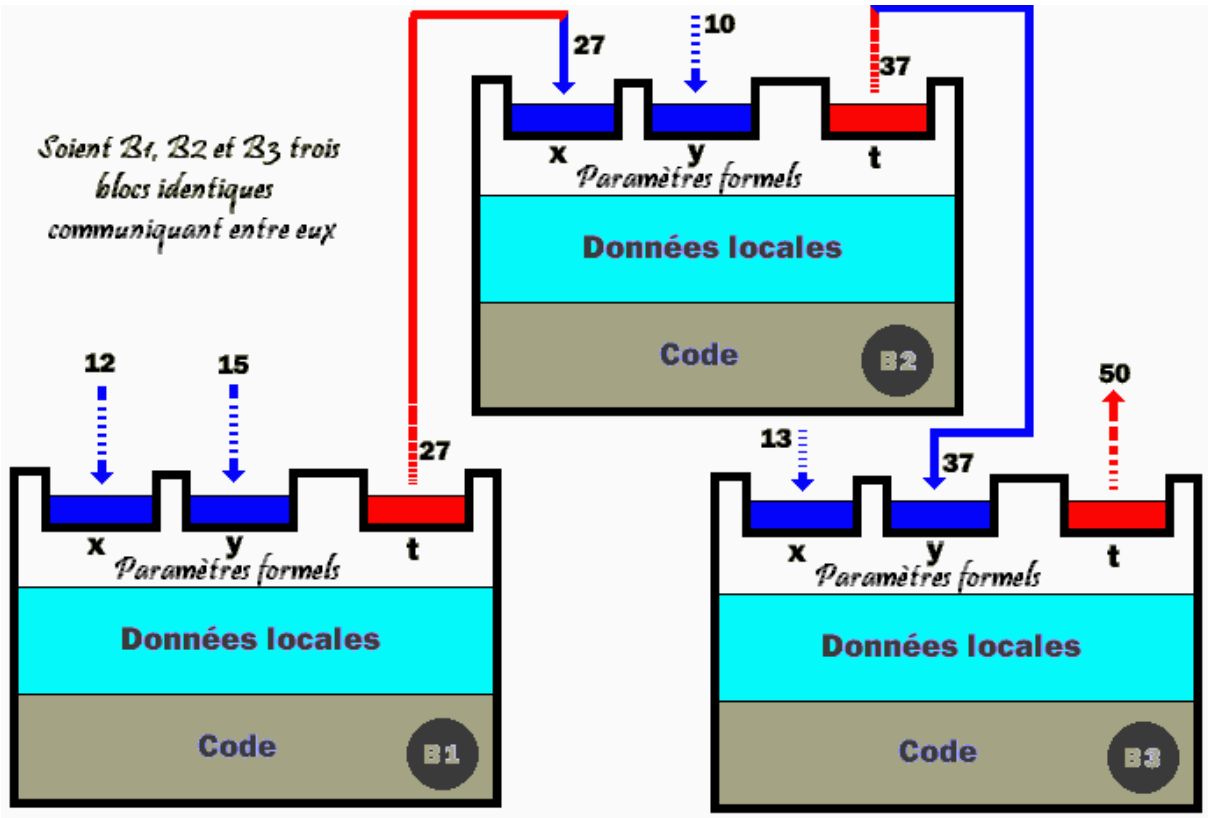
- Le paramètre effectif "**i**" est associé au paramètre formel **rang**.(**short** = entier signé sur 16 bits et **int** = entier signé sur 32 bits)

- Le paramètre effectif "**elt**" est associé au paramètre formel **val**.(**sbyte** = entier signé sur 8 bits et **long** = entier signé sur 64 bits)

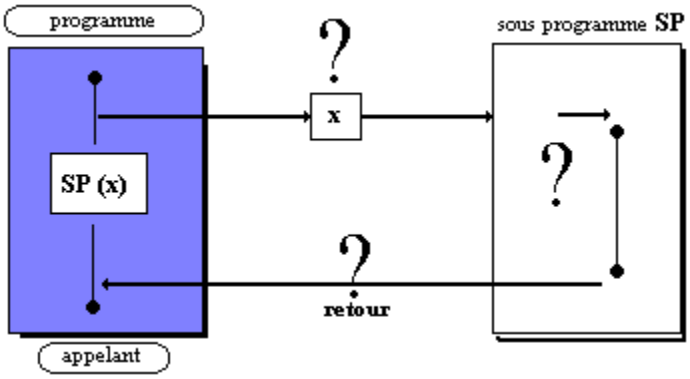
*Les **trois** modes principaux de transmission des paramètres sont :*

- passage par **valeur**
- passage par **référence**
- passage par **résultat**

Un paramètre effectif transmis au sous-programme appelé est en fait un moyen d'utiliser ou d'accéder à une information appartenant au bloc appelant (le bloc appelé peut être le même que le bloc appelant, il s'agit alors de récursivité).



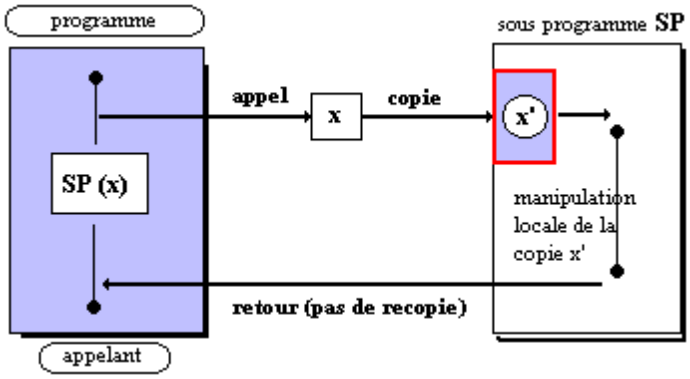
La question technique qui se pose en C# comme dans tout langage de programmation est de connaître le fonctionnement du passage des paramètres :



En C#, ces trois modes de transmission (ou de passage) des paramètres (très semblables à Delphi) sont implantés.

3.1 - Les paramètres C# passés par valeur

Le passage par **valeur** est valable pour tous les types élémentaires (**int**, **byte**, **short**, **long**, **bool**, **double, float**, **char**) et les objets.



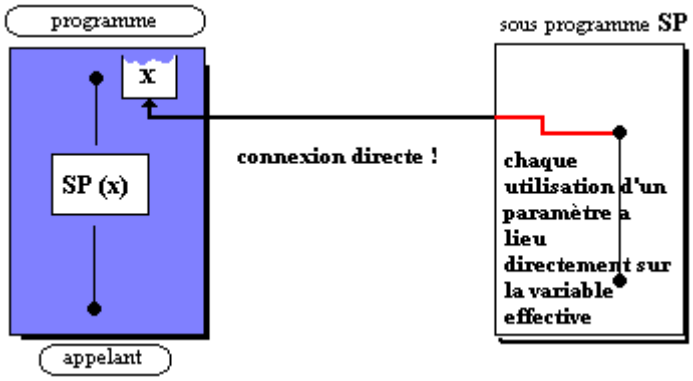
En C# tous les paramètres sont passés par défaut par valeur (lorsque le paramètre est un objet, c'est en fait **la référence de l'objet qui est passée par valeur**). Pour ce qui est de la vision algorithmique de C#, le passage par valeur permet à une variable d'être passée comme paramètre d'entrée.

```
static int methode1(int a , char b) {
 //.......
  return a+b;
}
```

Cette méthode possède 2 paramètres **a** et **b** en entrée passés par valeur et renvoie un résultat de type **int**.

3.2 - Les paramètres C# passés par référence

Le passage par **référence** est valable pour tous les types de C#.



En C# pour indiquer un passage par référence on précède la déclaration du paramètre formel du mot clef **ref :**

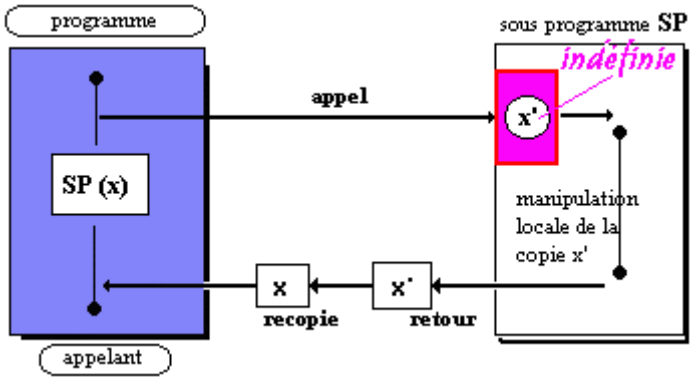
```
static int methode1(int a , ref char b) {
 //.......
  return a+b;   }
```

Lors de l'appel d'un paramètre passé par référence, le mot clef **ref** doit **obligatoirement** précéder le paramètre effectif qui doit obligatoirement avoir été initialisé auparavant :

```
int x = 10, y = '$', z = 30;
z = methode1(x, ref y) ;
```

3.3 - Les paramètres C# passés par résultat

Le passage par **résultat** est valable pour tous les types de C#.



En C# pour indiquer un passage par résultat on précède la déclaration du paramètre formel du mot

out :

```
static int methode1(int a , out char b) {
 //.......
  return a+b;
}
```

Lors de l'appel d'un paramètre passé par résultat, le mot clef **out** doit **obligatoirement** précéder le paramètre effectif qui n'a pas besoin d'avoir été initialisé :

```
int x = 10, y , z = 30;
z = methode1(x, out y) ;
```

Remarque :

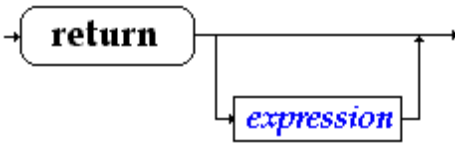
> Le choix de **passage selon les types** élimine les inconvénients dûs à l'encombrement mémoire et à la lenteur de recopie de la valeur du paramètre par exemple dans un passage par valeur, car nous verrons plus loin que les **tableaux en C# sont des objets** et que leur structure est passée **par référence**.

3.4 - Les retours de résultats de méthode type fonction

Les méthodes de type fonction en C#, peuvent renvoyer un résultat de n'importe quel type et acceptent des paramètres de tout type.

Une méthode- fonction ne peut renvoyer qu'**un seul** résultat comme en Java, mais l'utilisation des passages de paramètres par **référence** ou par **résultat**, permet aussi d'utiliser les paramètres de la fonction comme des variables de résultats comme en Delphi.

En C# comme en Java le retour de résultat est passé grâce au mot clef **return** placé n'importe où dans le corps de la méthode.

Syntaxe : 	L'expression lorsqu'elle est présente est quelconque mais doit être obligatoirement du même type que le type du résultat déclaré dans l'en-tête de fonction (ou d'un type compatible). Lorsque le **return** est rencontré il y a arrêt de l'exécution du code de la méthode et retour du résultat dans le bloc appelant.

4 - Visibilités des variables

Le principe de base est que les variables en C# sont visibles (donc utilisables) dans le bloc dans lequel elles ont été définies.

Visibilité de bloc

C# est un langage à structure de blocs (comme pascal et C) dont le principe général de visibilité est :

Toute variable déclarée dans un bloc est visible dans ce bloc et dans tous les blocs imbriqués dans ce bloc.

En C# les blocs sont constitués par :

- **les classes,**
- **les méthodes,**
- **les instructions composées,**
- **les corps de boucles,**
- **les try...catch**

Le masquage des variables n'existe que pour les variables déclarées dans des méthodes :

Il est **interdit de redéfinir** une variable déjà déclarée dans une méthode soit :

comme paramètre de la méthode,

comme variable locale à la méthode,

dans un bloc inclus dans la méthode.

Il est **possible de redéfinir** une variable déjà déclarée dans une classe.

Variables dans une classe, dans une méthode

Les variables définies (déclarées, et/ou initialisées) dans une classe sont accessibles à toutes les méthodes de la classe, la visibilité peut être modifiée par les qualificateurs **public** ou **private** que nous verrons au chapitre C# et POO.

Exemple de visibilité dans une classe

```
class ExempleVisible1 {
 int a = 10;

      int g (int x )
      {  return  3*x-a;
      }

      int f (int x, int a )
      {  return 3*x-a;
      }

 }
```

La variable "a" définie dans **int a** =10; :

- Est une variable de la classe ExempleVisible.

- Elle est visible dans la méthode **g** et dans la méthode **f**. C'est elle qui est utilisée dans la méthode **g** pour évaluer l'expression 3*x-**a.**

- Dans la méthode **f**, elle est masquée par le paramètre du même nom qui est utilisé pour évaluer l'expression 3*x-**a.**

Contrairement à ce que nous avions signalé plus haut nous n'avons pas présenté un exemple fonctionnant sur des méthodes de classes (qui doivent obligatoirement être précédées du mot clef **static**), mais sur des méthodes d'instances dont nous verrons le sens plus loin en POO.

Remarquons avant de présenter le même exemple cette fois-ci sur des méthodes de classes, que quelque soit le genre de méthode la visibilité des variables est **identique**.

Exemple identique sur des méthodes de classe

```
class ExempleVisible2 {
 static int a = 10;

      static int g (int x )
      {  return  3*x-a;
      }

      static int f (int x, int a )
      {  return 3*x-a;
      }

    }
```

La variable "a" définie dans **static int a** =10; :

- Est une variable de la classe ExempleVisible.

- Elle est visible dans la méthode **g** et dans la méthode **f**. C'est elle qui est utilisée dans la méthode **g** pour évaluer l'expression 3*x-**a.**

- Dans la méthode **f**, elle est masquée par le paramètre du même nom qui est utilisé pour évaluer l'expression 3*x-**a.**

Les variables définies dans une méthode (de classe ou d'instance) suivent les règles classiques de la visibilité du bloc dans lequel elles sont définies :

> Elles sont visibles dans toute la méthode et dans tous les blocs imbriqués dans cette méthode et seulement à ce niveau (les autres méthodes de la classe ne les voient pas), c'est pourquoi on emploie aussi le terme de variables locales à la méthode.

Reprenons l'exemple précédent en adjoignant des variables locales aux deux méthodes f et g.

Exemple de variables locales

```
class ExempleVisible3 {
 static int a = 10;

      static int g (int x )
      {  char car = 't';
         long a = 123456;
          ....
         return  3*x-a;
      }

      static int f (int x, int a )
      {  char car ='u';
          ....
         return 3*x-a;
      }

 }
```

La variable de classe "a" définie dans **static int a** = 10; est masquée dans les deux méthodes **f** et **g**.

Dans la méthode **g**, c'est la variable locale **long a** = 123456 qui masque la variable de classe **static int a**. **char** car ='t'; est une variable locale à la méthode **g**.

- Dans la méthode **f**, **char** car ='u'; est une variable locale à la méthode **f**, le paramètre **int a** masque la variable de classe **static int a**.

Les variables locales **char** car n'existent que dans la méthode où elles sont définies, les variables "car" de **f** et celle de **g** n'ont aucun rapport entre elles, bien que portant le même nom.

Variables dans un bloc autre qu'une classe ou une méthode

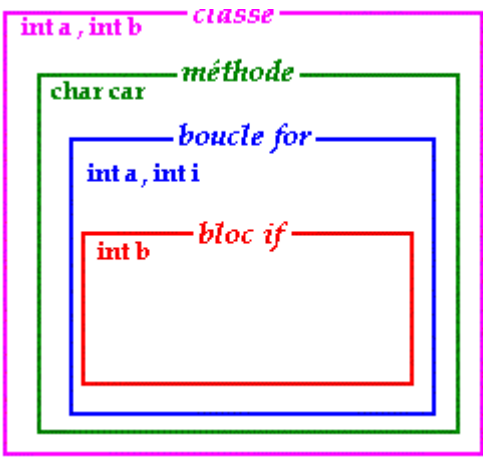
Les variables définies dans des blocs du genre instructions composées, boucles, **try…catch** ne sont visibles que dans le bloc et ses sous-blocs imbriqués, dans lequel elles sont définies.

Toutefois attention aux ***redéfinitions*** de variables locales. Les blocs du genre instructions composées, boucles, **try…catch** ne sont utilisés qu'à l'intérieur du corps d'une méthode (ce sont les actions qui dirigent le fonctionnement de la méthode), les variables définies dans de tels blocs sont automatiquement considérées par C# comme des variables locales à la méthode. Tout en respectant à l'intérieur d'une méthode le principe de visibilité de bloc, C# **n'**accepte **pas** le masquage de variable à l'**intérieur des blocs imbriqués**.

Nous donnons des exemples de cette visibilité :

Exemple correct de variables locales

```
class ExempleVisible4 {

   static int a = 10, b = 2;

    static int f (int x )

    {  char car = 't';

       for (int i = 0;  i < 5 ;  i++)
       {int a=7;

         if (a < 7)
          {int b = 8;
           b = 5-a+i*b;
          }

        else b = 5-a+i*b;
```



La variable de classe "a" définie dans **static int a** =

```
        }

        return  3*x-a+b;
    }
  }
```

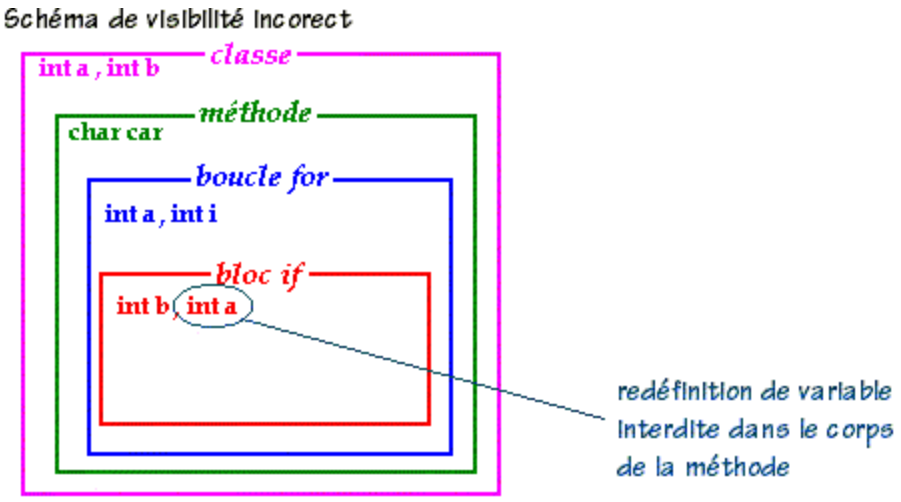
10; est masquée dans la méthode **f** dans le bloc imbriqué **for**.

La variable de classe "b" définie dans **static int b** = 2; est masquée dans la méthode **f** dans le bloc imbriqué **if**.

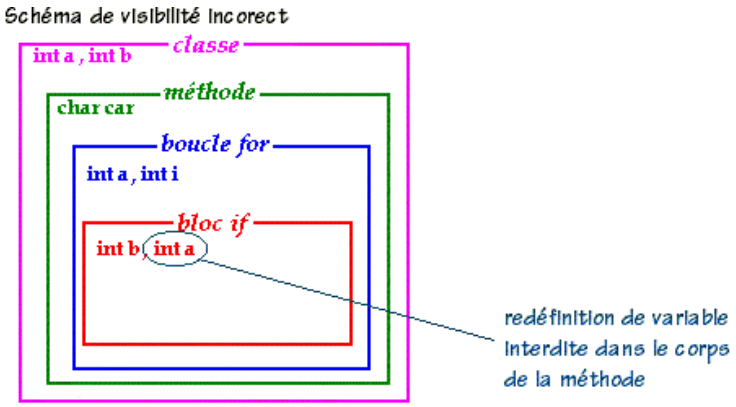
Dans l'instruction **{ int b** = 8; **b** = 5-**a**+**i***b; **}** , c'est la variable **b** interne à ce bloc qui est utilisée car elle masque la variable **b** de la classe.

Dans l'instruction **else b** = 5-**a**+**i***b; , c'est la variable **b** de la classe qui est utilisée (car la variable **int b** = 8 n'est plus visible ici) .

Exemple de variables locales générant une erreur




```
class ExempleVisible5 {

    static int a = 10, b = 2;

     static int f (int x )

     {  char car = 't';

        for (int i = 0;  i < 5 ;  i++)
        {int a=7;

          if (a < 7)

           {int b = 8, a = 9;
            b = 5-a+i*b;
           }

        else b = 5-a+i*b;
        }
```



Toutes les remarques précédentes restent valides puisque l'exemple ci-contre est quasiment identique au précédent. Nous avons seulement rajouté dans le bloc **if** la définition d'une nouvelle variable interne **a** à ce bloc.

C# produit une erreur de compilation **int b** = 8, **a = 9**; sur la variable **a**, en indiquant que c'est une ***redéfinition*** de

```
        return  3*x-a+b;
    }
  }
```

variable à l'intérieur de la méthode **f**, car nous avions déjà défini une variable **a** (**{ int a**=7;...) dans le bloc englobant **for {...}**.

Remarquons que le principe de visiblité des variables adopté en C# est identique au principe inclus dans tous les langages à structures de bloc y compris pour le **masquage**, s'y rajoute en C# comme en Java, uniquement l'interdiction de la **redéfinition** à l'intérieur d'une même méthode (semblable en fait, à l'interdiction de redéclaration sous le même nom, de variables locales à un bloc).

Spécificités du compilateur C#

Le compilateur C# n'accepte pas que l'on utilise une **variable de classe** qui a été **redéfinie dans un bloc** interne à une méthode, sans la qualifier par la notation uniforme aussi appelée opération de résolution de portée. Les 4 exemples ci-dessous situent le discours :

```
class ClA1 {
  static int a = 10, b = 2;
  static int f (int x )
  {  char car = 't';
     for (int i = 0;  i < 5 ;  i++)
     {int a=7;
       if (a < 7)
        {
         b = 5-a+i*b;
        }
        else b = 10-a+i*b;
     }
     return  3*x-a+b;   ← erreur
      return  3*x- ClA1.a+b;
  }
}
```

Variables de classe

'a' redéfinie

Refus 'a' : conflit

accepté

```
class ClA2 {
  static int a = 10, b = 2;
  static int f (int x )
  {  char car = 't';
     int b = 8;
     for (int i = 0;  i < 5 ;  i++)
     {int a=7;
       if (a < 7)
        {
         b = 5-a+i*b;
        }
        else b = 10-a+i*b;
     }
     return  3*x-a+b;   ← erreur
      return  3*x- ClA2.a+b;
  }
}
```

Variables de classe

' b' redéfinie

' a' redéfinie

Refus 'a' : conflit

accepté

```
class ClA3 {
  static int a = 10, b = 2;
  static int f (int x )
  {  char car = 't';
     for (int i = 0;  i < 5 ;  i++)
     {int a=7;
       if (a < 7)
        { int b = 8;
          b = 5-a+i*b;
        }
       else b = 10-a+i*b;
       else ClA3.b = 10-a+i* ClA3.b;
     }
     return  3*x-a+b;
     return  3*x- ClA3.a+ ClA3.b;
  }
}
```

Variables de classe

' a' redéfinie

' b' redéfinie

Refus 'b' : conflit

accepté

Refus 'a' et 'b' : conflit

accepté

```
class ClA4 {
  static int a = 10, b = 2;
  static int f (int x )
  {  char car = 't';
     for (int i = 0;  i < 5 ;  i++)
     {int a=7; int b = 8;
       if (a < 7)
        {
         b = 5-a+i*b;
        }
        else b = 10-a+i*b;
     }
     return  3*x-a+b;
     return  3*x- ClA4.a+ ClA4.b;
  }
}
```

Variables de classe

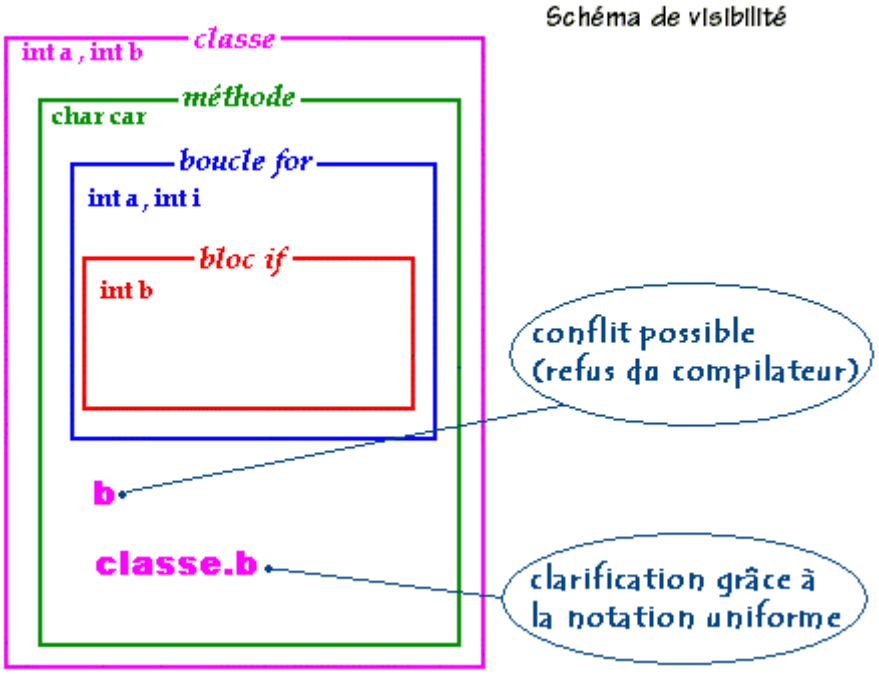
' a' et ' b' redéfinies

Refus 'a' et 'b' : conflit

accepté

Observez les utilisations et les redéfinitions correctes des variables "**static int a** = 10, **int b** = 2;" déclarées comme variables de classe et redéfinies dans différents blocs de la classe (lorsqu'il y a un conflit signalé par le compilateur sur une instruction nous avons mis tout de suite après le code correct accepté par le compilateur)

Là où le compilateur C# détecte un conflit potentiel, il suffit alors de qualifier la variable grâce à l'opérateur de résolution de portée, comme par exemple ClasseA.**b** pour indiquer au compilateur la variable que nous voulons utiliser.



Comparaison C# , java : la même classe à gauche passe en Java mais fournit 6 conflits en C#

En Java	Conflits levés en C#

class ClA5 {
static int a = 10, b = 2;
static int f (**int** x)
{ **char** car = 't';
for (**int** i = 0; i < 5 ; i++)
{
if (a < 7)
{ **int** b = 8 , a = 7;
b = 5-a+i*b;
}
else
b = 10-a+i* b;
}
return 3*x- a+ b;
}
}
	class ClA5 {
	static int a = 10, b = 2;
	static int f (**int** x)
	{ **char** car = 't';
	for (**int** i = 0; i < 5 ; i++)
	{
	if (ClA5.a < 7)
	{ **int** b = 8 , a = 7;
	b = 5-a+i*b;
	}
	else
	ClA5.b = 10-ClA5.a+i*ClA5.b;
	}
	return 3*x- ClA5.a+ ClA5.b;
	}
	}

' a' et ' b' redéfinies

Evite le conflit

Les chaînes de caractères string

La classe string

Le type de données String (chaîne de caractère) est une classe de **type référence** dans l'espace de noms **System** de .NET Framework. Donc une chaîne de type **string** est un objet qui n'est utilisable qu'à travers les méthodes de la classe **string**. Le type **string** est un alias du type System.**String** dans .NET

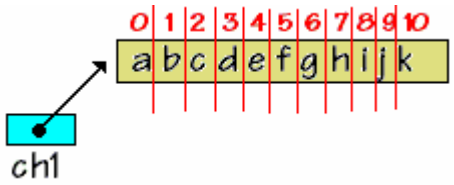
> Un littéral de chaîne est une suite de caractères entre guillemets : " abcdef " est un exemple de littéral de String.

Toutefois un objet **string** de C# est immuable (son contenu ne change pas)

- Etant donné que cette classe est très utilisée les variables de type string bénéficient d'un statut d'utilisation aussi souple que celui des autres **types élémentaires par valeurs**. On peut les considérer comme des listes de caractères Unicode numérotés de 0 à n-1 (si n figure le nombre de caractères de la chaîne).
- Il est possible d'accéder à chaque caractère de la chaîne en la considérant comme un **tableau de caractères en lecture seule**.

Accès à une chaîne string

Déclaration d'une variable String	**String** str1;
Déclaration d'une variable String avec initialisation	**String** str1 = " abcdef "; Instancie un nouvel objet : "abcdef "
On accède à la longueur d'une chaîne par la propriétée : **int** Length	**String** str1 = "abcdef"; **int** longueur; longueur = str1.Length ; *// ici longueur = 6*

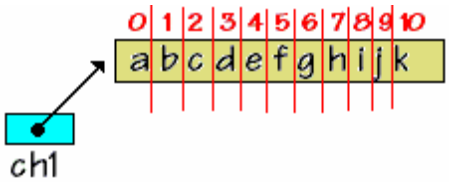
On accède à un caractère de rang fixé d'une chaîne par l'opérateur [ ] : (*la chaîne est lue comme un tableau de char*) Il est possible d'accéder en lecture seulement à chaque caractères d'une chaîne, mais qu'il est impossible de modifier un caractère directement dans une chaîne.	**String** ch1 = "abcdefghijk"; Représentation interne de l'objet ch1 de type **string** :  **char** car = ch1**[**4**]** ; *// ici la variable car contient la lettre 'e'*

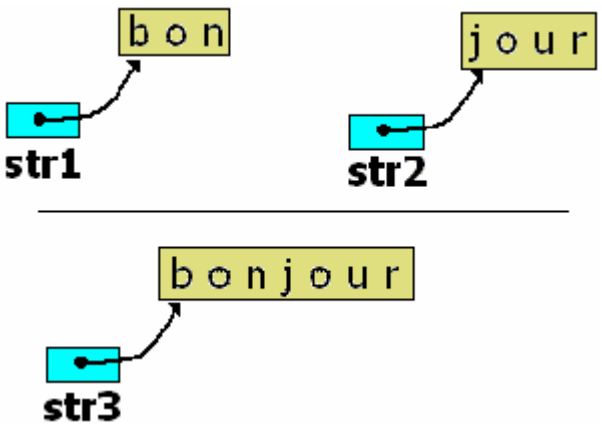
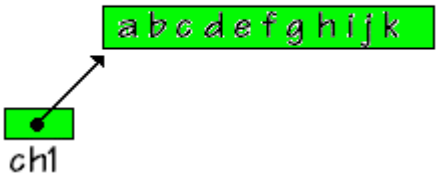
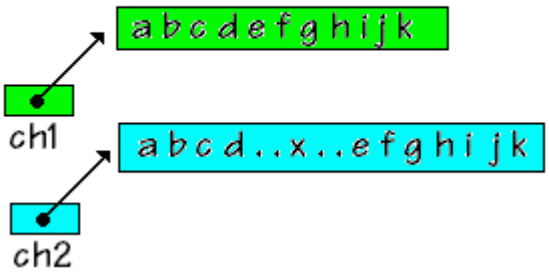
Remarque

> En fait l'opérateur [ ] est un indexeur de la classe **string** (cf. chapitre indexeurs en C#), et il est en lecture seule :
> **public char this** [**int** index] { **get** ; }
>
> Ce qui signifie au stade actuel de compréhension de C#, qu'il est possible d'accéder en lecture seulement à chaque caractère d'une chaîne, mais qu'il est impossible de modifier un caractère grâce à l'indexeur.
>
> **char** car = ch1[7] ; *// l'indexeur renvoie le caractère 'h' dans la variable car.*
> ch1[5] = car ; *// Erreur de compilation : l'écriture dans l'indexeur est interdite !!*
> ch1[5] = 'x' ; *// Erreur de compilation : l'écriture dans l'indexeur est interdite !!*

Opérations de base sur une chaîne string

Le type **string** possède des méthodes d'**insertion**, **modification** et **suppression :** méthodes **Insert**, **Copy**, **Concat**,...

Position d'une sous-chaîne à l'intérieur d'une chaîne donnée : **Méthode surchargée 6 fois :** **int** IndexOf (**…**) Ci-contre une utilisation de la surcharge : **int** IndexOf (**string** ssch) qui renvoie l'indice de la première occurrence du **string** ssch contenue dans la chaîne scannée.	**Recherche de la position de ssch dans ch1 :** **String** ch1 = " abcdef " , ssch="cde";  **int** rang ; rang = ch1.IndexOf (ssch); *// ici la variable rang vaut 2*

Concaténation de deux chaînes **Un opérateur ou une méthode** **Opérateur : + sur les chaînes** ou **Méthode static surchargée 8 fois** : **String** Concat(**…**) Les deux écritures ci-dessous sont donc équivalentes en C# : str3 = str1+str2 ⇔ str3 = str1.Concat(str2)	**String** str1,str2,str3; str1="bon"; str2="jour"; str3=str1+str2; 
Insertion d'une chaîne **dans** **une autre chaîne** Appel de la méthode **Insert** de la chaîne ch1 afin de construire une nouvelle chaîne ch2 qui est une copie de ch1 dans laquelle on a inséré une sous-chaîne, ch1 n'a pas changé (immuabilité).	Soit : ch1 = "abcdefghijk" ;  Puis invocation de la méthode Insert sur ch1 : ch2 = ch1.Insert(4,"..x..") ;  **ch2** est une copie de **ch1** dans laquelle on a insérer la sous-chaîne "**..x..**".

Attention :

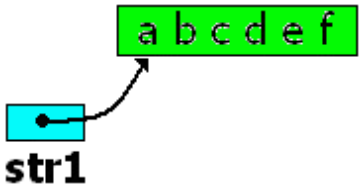
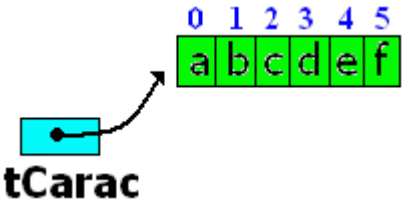
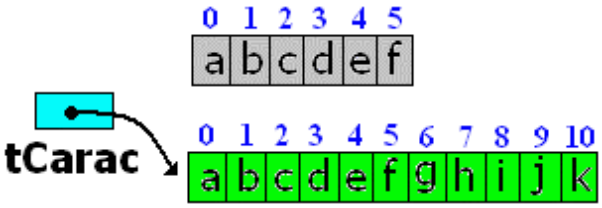
> Les méthodes d'insertion, suppression, etc…ne modifient pas la chaîne objet qui invoque la méthode mais renvoient un autre objet de chaîne différent, obtenu après action de la méthode sur l'objet initial.

Convertir une chaîne string en tableau de caractères

Si l'on souhaite se servir d'une **string** comme un tableau de **char**, il faut utiliser la méthode **ToCharArray** qui convertit la chaîne en un tableau de caractères contenant tous les caractères de la chaîne.

Soient les lignes de programme suivantes :
string str1 = "abcdef" ;
char [ ] tCarac ;
tCarac = str1.**ToCharArray**( ) ;
tCarac = "abcdefghijk".ToCharArray( );

Illustrons ces lignes par des schémas de références :

string str1 = "abcdef" ;	
char [ ] tCarac ; tCarac = str1.**ToCharArray**( ) ;	
tCarac = "abcdefghijk".ToCharArray( );	

Opérateurs d'égalité et d'inégalité de string

L'opérateur d'égalité = = , détermine si deux objets **string** spécifiés ont la **même valeur**, il se comporte comme sur des éléments de type de base (int, char,...)

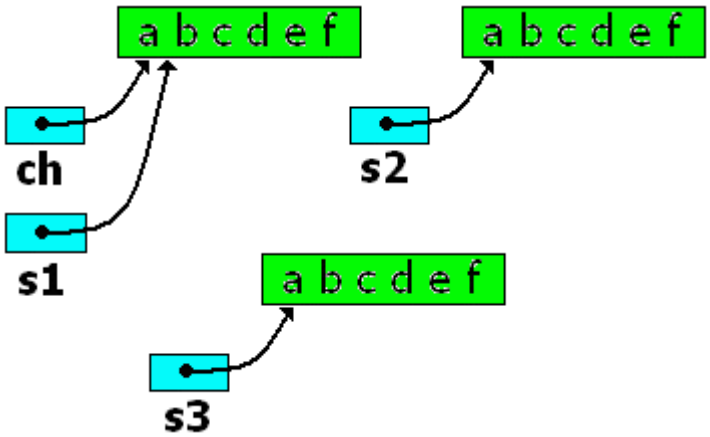
string a , b ;
(a = = b) renvoie **true** si la valeur de a est la même que la valeur de b ; sinon il renvoie **false**.

public static bool operator == (**string** a, **string** b);

Cet opérateur est surchargé et donc il compare les valeurs effectives des chaînes et non leur références, il fonctionne comme la méthode **public** bool Equals(**string** value) de la classe **string**, qui teste l'égalité de valeur de deux chaînes.

Voici un morceau de programme qui permet de tester l'opérateur d'égalité = = et la méthode Equals :

```
string s1,s2,s3,ch;
ch = "abcdef";
s1 = ch;
s2 = "abcdef";
s3 = new string("abcdef".ToCharArray( ));
```




```
System.Console.WriteLine("s1="+s1);
System.Console.WriteLine("s2="+s2);
System.Console.WriteLine("s3="+s3);
System.Console.WriteLine("ch="+ch);
if( s2 == ch )System.Console.WriteLine("s2=ch");
  else System.Console.WriteLine("s2<>ch");
if( s2 == s3 )System.Console.WriteLine("s2=s3");
  else System.Console.WriteLine("s2<>s3");
if( s3 == ch )System.Console.WriteLine("s3=ch");
  else System.Console.WriteLine("s3<>ch");
if( s3.Equals(ch) )System.Console.WriteLine("s3 égal ch");
  else System.Console.WriteLine("s3 différent de ch");
```

Après exécution on obtient :

s1=abcdef
s2=abcdef
s3=abcdef
ch=abcdef
s2=ch
s2=s3
s3=ch
s3 égal ch

POUR LES HABITUES DE JAVA : ATTENTION

L'opérateur d'égalité == en Java (Jdk1.4.2) n'est pas surchargé, il ne fonctionne pas totalement de la même façon que l'opérateur == en C#, car il ne compare que les références. Donc des programmes en apparence syntaxiquement identiques dans les deux langages, peuvent produire des résultats d'exécution différents :

Programme Java	Programme C#
String ch; ch = "abcdef" ; **String** s2,s1="abc" ; s2 = s1+"def"; *//-- tests d'égalité avec l'opérateur = =* **if**(s2 == "abcdef") System.out.println ("s2==abcdef"); **else** System.out.println ("s2<>abcdef"); **if**(s2 == ch) System.out.println ("s2==ch"); **else** System.out.println ("s2<>ch");	**string** ch; ch = "abcdef" ; **string** s2,s1="abc" ; s2 = s1+"def"; *//-- tests d'égalité avec l'opérateur = =* **if**(s2 == "abcdef") System.Console.WriteLine ("s2==abcdef"); **else** System.Console.WriteLine ("s2<>abcdef"); **if**(s2 == ch) System.Console.WriteLine ("s2==ch"); **else** System.Console.WriteLine ("s2<>ch");
Résultats d'exécution du code Java : **s2<>abcdef** **s2<>ch**	**Résultats d'exécution du code C# :** **s2==abcdef** **s2==ch**

Rapport entre string et char

Une chaîne **string** contient des éléments de base de type **char,** comment passe-t-on de l'un à l'autre type ?

1°) On ne peut pas considérer un **char** comme un cas particulier de **string**, le transtypage suivant est refusé comme en Java :

```
char car = 'r';
string s;

s = (string)car;
```

Il faut utiliser l'une des surcharges de la méthode de conversion ToString de la classe **Convert** :

System.Object

|__**System.Convert**

méthode de classe static :

public static string ToString**(** char ***c*** **);**

Le code suivant est correct, il permet de stocker un caractère **char** dans une **string** :

```
char car = 'r';
string s;

s = Convert.ToString (car);
```

Remarque :

> La classe **Convert** contient un grand nombre de méthodes de conversion de types. Microsoft indique que cette classe : "constitue une façon, indépendante du langage, d'effectuer les conversions et est disponible pour tous les langages qui ciblent le **C**ommon **L**anguage **R**untime. Alors que divers langages peuvent recourir à différentes techniques pour la conversion des types de données, la classe **Convert** assure que toutes les conversions communes sont disponibles dans un format générique."

2°) On peut concaténer avec l'opérateur +, des **char** à une chaîne **string** déjà existante et affecter le résultat à une String :

string s1 , s2 ="abc" ;

char c = 'e' ;

s1 = s2 + 'd' ;

s1 = s2 + c ;

Toutes les écritures précédentes sont licites et acceptées par le compilateur C#, il n'en est pas de même pour les écritures ci-après :

Les écritures suivantes seront refusées :	**Ecritures correctes associées :**
String s1 , s2 ="abc" ; **char** c = 'e' ; s1 = 'd' + c ; *// types incompatibles*	**String** s1 , s2 ="abc" ; **char** c = 'e' ; s1 = "d" + **Convert**.ToString (c) ;
s1 = 'd' + 'e'; *// types incompatibles*	s1 = "d" + "e"; s1 = "d" + 'e'; s1 = 'd' + "e";

Le compilateur enverra le message d'erreur suivant pour l'instruction `s1 = 'd' + c ;` et pour l'instruction `s1 = 'd' + 'e';` :

*[C# Erreur] : Impossible de convertir implicitement le type **'int'** en **'string'***

Car il faut qu'au moins un des deux opérandes de l'opérateur **+** soit du type **string** :

- Le littéral 'e' est de type **char**,
- Le littéral "e" est de type **string** (chaîne ne contenant qu'un seul caractère)

Pour plus d'information sur toutes les méthodes de la classe **string** consulter la documentation de .Net framework.

Tableaux et matrices

Généralités sur les tableaux

Dès que l'on travaille avec de nombreuses données homogènes (de même type) la première structure de base permettant le regroupement de ces données est le **tableau.** C# comme tous les langages algorithmiques propose cette structure au programmeur. Comme pour les string et pour des raisons d'efficacité dans l'encombrement mémoire, les tableaux sont gérés par C# , comme des objets de **type référence** (donc sur le tas), leur type hérite de la classe abstraite **System.Array**..

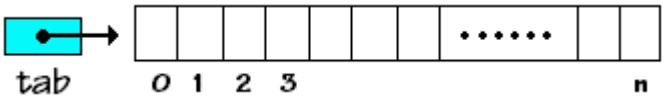
Les tableaux C# sont indexés uniquement par des entiers (**char**, **int**, **long**,…) et sur un intervalle fixe à partir de zéro. Un tableau C# peut avoir de une ou à plusieurs dimensions, nous avons donc les variétés suivantes de tableaux dans le CLR :

Tableaux à une dimension.
Tableaux à plusieurs dimensions (matrices,…) comme en Delphi.
Tableaux déchiquetés comme en Java.

Chaque dimension d'un tableau en C# est définie par une valeur ou longueur, qui est un nombre ou une variable **N** entier (**char**, **int**, **long**,…) dont la valeur est supérieur ou égal à zéro. Lorsqu'une dimension a une longueur **N**, l'indice associé varie dans l'intervalle [0 , **N** – 1].

Tableau uni-dimensionnel

Ci-dessous un tableau '**tab**' à une dimension, de n+1 cellules numérotées de 0 à n :



- Les tableaux C# contiennent comme en Delphi, des objets de types quelconques de C# (type **référence** ou type **valeur**).
- Il n'y a pas de mot clef spécifique pour la classe tableaux, mais l'opérateur symbolique **[ ]** indique qu'une variable de type fixé est un tableau.
- La taille d'un tableau doit obligatoirement avoir été définie avant que C# accepte que vous l'utilisiez !

Remarque :

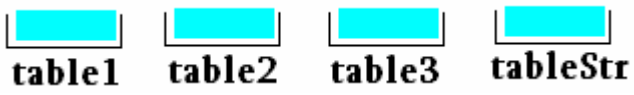
Les tableaux de C# sont des objets d'une classe dénommée **Array** qui est la classe de base d'implémentation des tableaux dans le CLR de .NET framework (localisation : **System.Array**) Cette classe n'est pas dérivable pour l'utilisateur : "**public abstract class** Array : ICloneable, IList, ICollection, IEnumerable".

C'est en fait le compilateur qui est autorisé à implémenter une classe physique de tableau. Il faut utiliser les tableaux selon la démarche ci-dessous en sachant que l'on dispose en plus des propriétés et des méthodes de la classe Array si nécessaire (longueur, tri, etc...)

Déclaration d'une variable de tableau, référence seule:

```
int [ ] table1;
char [ ] table2;
float [ ] table3;
...
string [ ] tableStr;
```

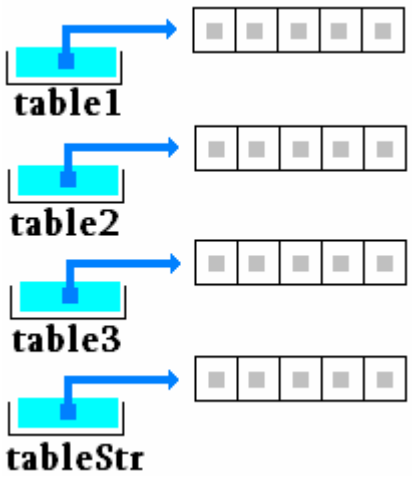
Voici une image de ce qui est présent en mémoire centrale après ces déclarations :



Déclaration d'une variable de tableau avec définition explicite de taille :

```
int [ ] table1 = new int [5];
char [ ] table2  = new char [5];
float [ ] table3  = new float [5];
...
string [ ] tableStr  = new String [5];
```

Voici une image de ce qui est présent en mémoire centrale après ces déclarations :



Chaque variable de reférence pointe vers un tableau de longueur 5, chaque cellule d'un tableau contient la valeur **null**.

Le mot clef **new** correspond à la **création d'un nouvel objet** (un nouveau tableau) dont la taille est fixée par la valeur indiquée entre les crochets. Ici 4 tableaux sont crées et prêts à être utilisés : table1 contiendra 5 entiers 32 bits, table2 contiendra 5 caractères, table3 contiendra 5 réels en simple précision et tableStr contiendra 5 chaînes de type string.

On peut aussi déclarer un tableau sous la forme de deux instructions : une instruction de déclaration et une instruction de définition de taille avec le mot clef **new**, la seconde pouvant être mise n'importe où dans le corps d'instruction, mais elle doit être utilisée avant toute manipulation du tableau. Cette dernière instruction de définition peut être répétée plusieurs fois dans le programme, il s'agira alors à chaque fois de la **création d'un nouvel objet** (donc un nouveau tableau), l'**ancien étant détruit** et désalloué automatiquement par le ramasse-miettes (garbage collector) de C#.

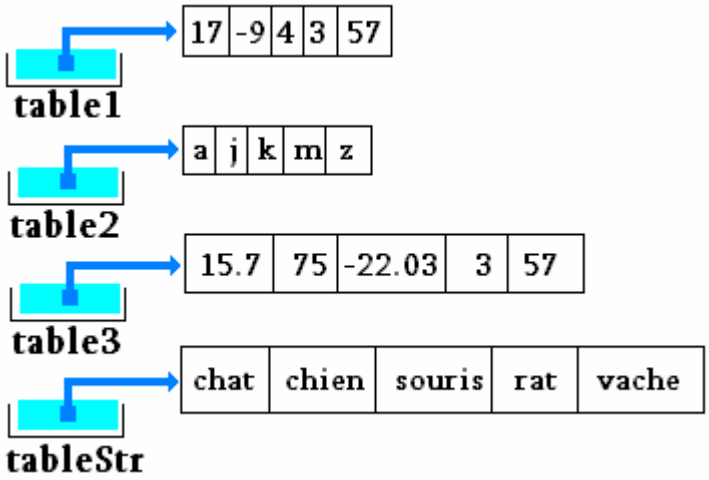
```
int [ ] table1;
char [ ] table2;
float [ ] table3;
string [ ] tableStr;
....
table1 = new int [5];
table2  = new char [12];
table3  = new float [8];
tableStr  = new string [9];
```

Déclaration et initialisation avec définition implicite de taille :

```
int [ ] table1 = {17,-9,4,3,57};
char [ ] table2  = {'a','j','k','m','z'};
float [ ] table3  = {-15.7f, 75, -22.03f, 3 ,57 };
string [ ] tableStr  = {"chat","chien","souris","rat","vache"};
```

Dans cette éventualité C# crée le tableau, calcule sa taille et l'initialise avec les valeurs fournies.

Voici une image de ce qui est présent en mémoire centrale après ces instructions :



Il existe en C# un attribut de la classe abstraite mère **Array**, qui contient **la taill**e d'un tableau uni-dimensionnel, quelque soit son type, c'est la propriété **Length** en lecture seule.

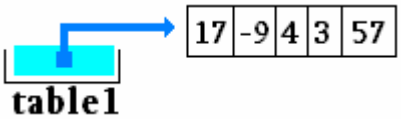
Exemple :

```
int [ ] table1 = {17,-9, 4, 3, 57};
int taille;
taille = table1.Length; // taille = 5
```

Attention

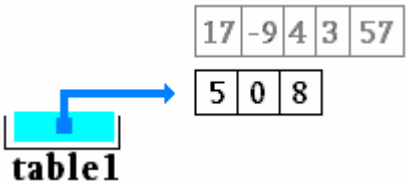
Il est possible de déclarer une référence de tableau, puis de l'initialiser après uniquement ainsi :

```
int [ ] table1 ; // crée une référence table1 de type tableau de type int
table1 = new int {17,-9, 4, 3, 57};  // instancie un tableau de taille 5 éléments referencé par table1
```



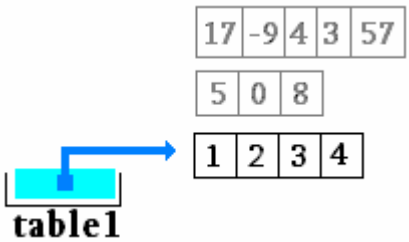
…

```
table1 = new int {5, 0, 8}; // instancie un autre tableau de taille 3 éléments referencé par table1
```



…

```
table1 = new int {1, 2, 3, 4}; // instancie un autre tableau de taille 3 éléments referencé par table1
```



Les cellules du tableau précédent sont perdues dès que l'on instancie table1 vers un nouveau tableau, leurs emplacements en mémoire centrale sont rendus au système de gestion de la mémoire (Garbage Collector).

L'écriture ci-dessous engendre une erreur à la compilation :

```
int [ ] table1 ; // crée une référence table1 de type tableau de type int
table1 =  {17,-9,4,3,57};  // ERREUR de compilation, correction :  int [ ] table1 = {17,-9,4,3,57};
```

Utiliser un tableau

Un tableau en C# comme dans les autres langages algorithmiques s'utilise à travers une cellule de ce tableau repérée par un indice obligatoirement de type entier ou un char considéré comme un entier (byte, short, int, long ou char). Le premier élément d'un tableau est numéroté **0**, le dernier **Length-1**.

On peut ranger des valeurs ou des expressions du type général du tableau dans une cellule du tableau.

Exemple avec un tableau de type **int** :

```
int [ ] table1 = new int [5];
// dans une instruction d'affectation:
table1[0] = -458;
table1[4] = 5891;
table1[5] = 72;  <--- est une erreur de dépassement de la taille ! (valeur entre 0 et 4)

// dans une instruction de boucle:
for (int i = 0 ; i<= table1.Length-1;  i++)
 table1[i] = 3*i-1; // après la boucle: table1 = {-1,2,5,8,11}
```

Même exemple avec un tableau de type **char** :

```
char [ ] table2  = new char [7];

table2[0] = '?' ;
table2[4] = 'a' ;
table2[14] = '#' ;  <--- est une erreur de dépassement de la taille
for (int i = 0 ; i<= table2.Length-1;  i++)
     table2[i] =(char)('a'+i);
// après la boucle: table2 = {'a', 'b', 'c' ,'d', 'e', 'f'}
```

Remarque :

> Dans une classe exécutable la méthode **Main** reçoit en paramètre un tableau de string nommé args qui correspond en fait aux éventuels paramètres de l'application elle-même:
>
> **static void Main**(string [ ] args)

Affectation de tableaux, recopie de tableaux uni-dimensionnels

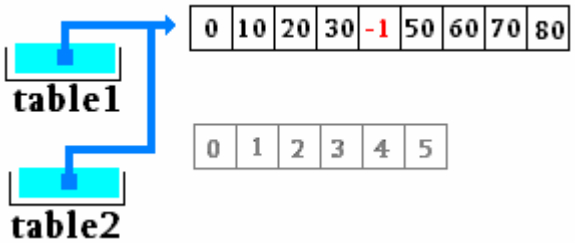
Un tableau est une reférence pointant vers un ensemble de cellules en mémoire centrale (c'est un

objet), il est tout à fait possible d'effectuer une affectation entre deux variables de tableaux de même type mais de taille égale ou différente, il ne s'agit alors pas d'une recopie de tableaux : ce sont en fait les pointeurs qui sont recopiés l'un dans l'autre par l'affectation et non pas les tableaux eux-mêmes.

Instruction	Image en mémoire centrale
int [ ]table1 = **new int** [9]; **int** [ ]table2 = **new int** [6];	table1 table2
table2 = table1	table1 table2 Après l'affectation : **table1** et **table2** pointent vers le même ensemble de cellules (le même tableau)

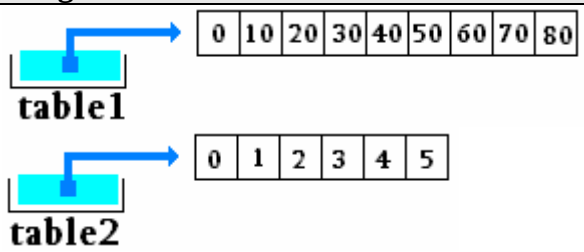
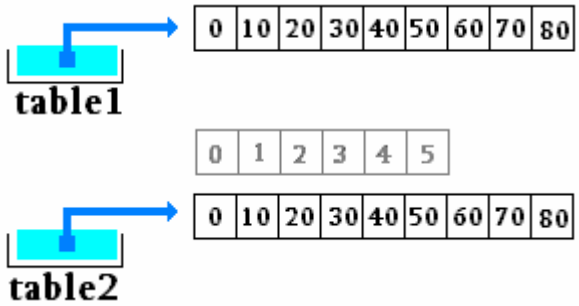
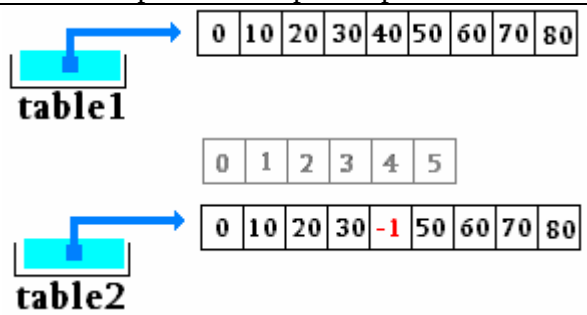
Toute modification sur table1 ou table2 a lieu sur les mêmes cellules :

Instructions	Image en mémoire centrale
int [ ]table1 = **new int** [9]; **int** [ ]table2 = **new int** [6];	table1 table2
for(**int** k = 0; k<=8; k++) table1(k) = k * 10 ;	table1 0 10 20 30 40 50 60 70 80
for(**int** k = 0; k<=5; k++) table2(k) = k ;	table2 0 1 2 3 4 5
table2 = table1;	table1 table2 0 10 20 30 40 50 60 70 80 0 1 2 3 4 5 table1 et table2 maintenant se refèrent à la même entité.

table2(4) = -1;	 Donc table1(4) vaut aussi "-1", car il pointe vers le même tableau qui ne se trouve qu'en un seul exemplaire en mémoire centrale.

Si l'on veut recopier un tableau uni-dimensionnel et en avoir deux exemplaires en mémoire, l'affectation ne le permet pas, il faut alors utiliser la méthode CopyTo(…) de la classe Array en lui passant comme paramètre le tableau de destination et en démarrant l'index de copie à la valeur 0.

En reprenant les tableaux précédents :

Instruction	Image en mémoire centrale
int [ ]table1 = **new int** [9]; **int** [ ]table2 = **new int** [6]; **for**(**int** k = 0; k<=8; k++) table1(k) = k * 10; **for**(**int** k = 0; k<=5; k++) table2(k) = k ;	
table1.CopyTo(table2,0);	 Cette instruction crée un nouveau tableau pointé par table2 qui l'exacte copie de celui pointé par table1.
table2(4) = -1;	 La modification de la cellule de rang 4 dans table2 n'a aucune influence sur table1 puisque ce sont deux entités bien distinctes.

Les matrices et les tableaux multi-dimensionnels

Les tableaux C# peuvent avoir plusieurs dimensions, ceux qui ont deux dimensions sont dénommés matrices (vocabulaire scientifique). Tous ce que nous allons dire sur les matrices s'étend ipso facto aux tableaux de dimensions trois, quatre et plus. Ce sont aussi des objets et ils se comportent comme les tableaux à une dimension tant au niveau des déclarations qu'au niveau des utilisations. La déclaration s'effectue avec un opérateur crochet et des virgules, exemples d'une syntaxe de déclaration d'un tableau à trois dimension : **[, ,]** . Leur structuration est semblable à celle des tableaux Delphi-pascal.

Déclaration d'une matrice, référence seule:

```
int [ , ] table1;
char [ , ] table2;
float [ , ] table3;
...
string [ , ] tableStr;
```

Déclaration d'une matrice avec définition explicite de taille :

```
int [ , ] table1 = new int [5, 2];
char [ , ] table2  = new char [9,4];
float [ , ] table3  = new float [2;8];
...
string [ , ] tableStr  = new String [3,9];
```

Exemple d'écriture de matrices de type int :

```
int [ , ] table1 = new int [2 , 3 ];// deux lignes de dimension 3 chacunes


// dans une instruction d'affectation:
table1[ 0 , 0 ] = -458;
table1[ 2 , 5 ] = -3;  <--- est une erreur de dépassement ! (valeur entre 0 et 1)
table1[ 1 , 4 ] = 83;  <--- est une erreur de dépassement ! (valeur entre 0 et 2)

// dans une instruction de boucle:
for (int i = 0 ; i<= 2;  i++)
   table1[1 , i ] = 3*i-1;


 // avec initialisation d'une variable dans la déclaration :
int n ;
int [ , ] table1 = new int [4 , n =3 ];// quatre lignes de dimension 3 chacunes
```

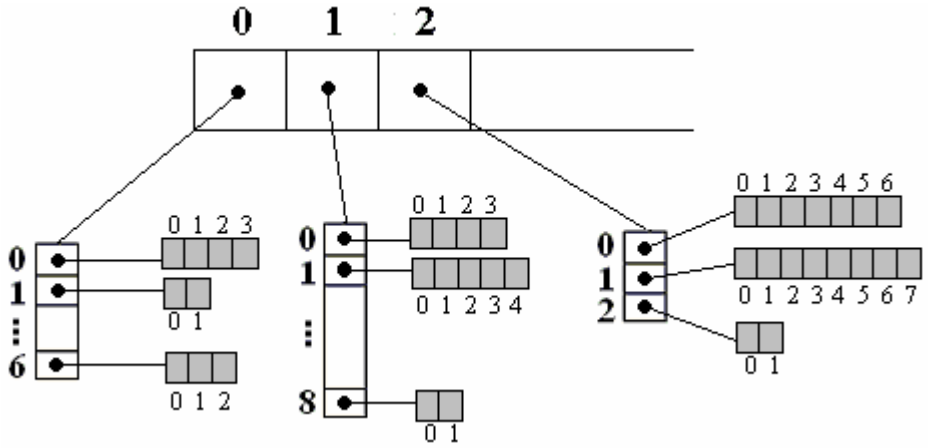
L'attribut **Length** en lecture seule, de la classe abstraite mère **Array**, contient en fait **la taille** d'un tableau en nombre total de cellules qui constituent le tableau (nous avons vu dans le cas uni-dimensionnel que cette valeur correspond à la taille de la dimension du tableau). Dans le cas d'un tableau multi-dimensionnel **Length** correspond au produit des tailles de chaque dimension d'indice :

int [,] table1 = **new int** [5, 2]; ⟶ table1.**Length** = 5 x 2 = 10
char [, ,] table2 = **new char** [9,4,5]; ⟶ table2.**Length** = 9 x 4 x 5 = 180
float [, , ,] table3 = **new float** [2,8,3,4]; ⟶ table3.**Length** = 2 x 8 x 3 x 4= 192

Tableaux déchiquetés ou en escalier

Leur structuration est strictement semblable à celle des tableaux Java, en fait en C# un tableau déchiqueté est composé de plusieurs tableaux unidimensionnels de taille variable. La déclaration s'effectue avec des opérateurs crochets **[ ] [ ]…** : autant de crochets que de dimensions. **C# autorise comme Java, des tailles différentes pour chacun des sous-tableaux.**

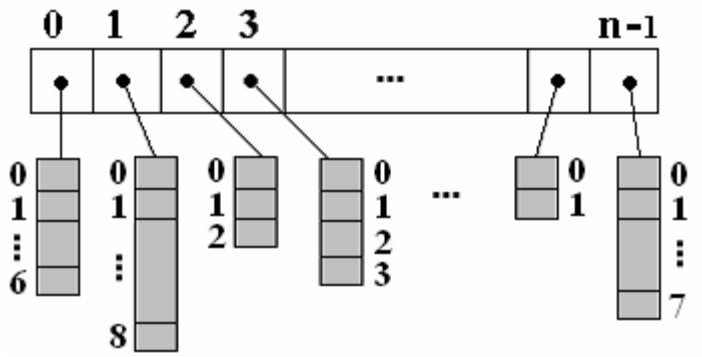
Ci-dessous le schéma d'un tableau T à trois dimensions en escalier :



Ce schéma montre bien qu'un tel tableau T est constitué de tableaux unidimensionnels, les tableaux composés de cases blanches contiennent des pointeurs (références). Chaque case blanche est une référence vers un autre tableau unidimensionnel, seules les cases grisées contiennent les informations utiles de la structure de données : les éléments de même type du tableau T.

Pour fixer les idées figurons la syntaxe des déclarations en C# d'un tableau d'éléments de type **int** nommé **myArray** bi-dimensionnel en escalier :

```
int n = 10;
int [ ][ ] myArray = new int [n][ ];
myArray[0] =  new int[7];
myArray[1] =  new int[9];
myArray[2] =  new int[3];
myArray[3] =  new int[4];
…
myArray[n-2] =  new int[2];
myArray[n-1] =  new int[8];
```

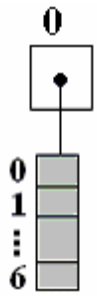
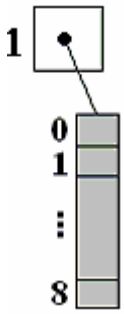


Ce tableau comporte 7+9+3+4+….+2+8 cellules utiles au stockage de données de type **int**, on peut le considérer comme une succession de tableaux d'int unidimensionnels (le premier ayant 7 cellules, le second ayant 9 cellules, etc…) . Les déclarations suivantes :

int n = 10;

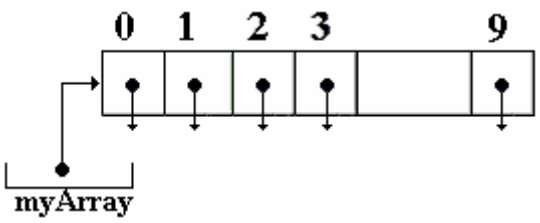
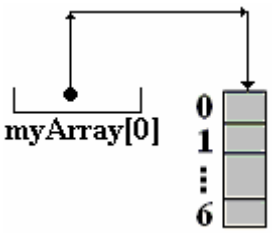
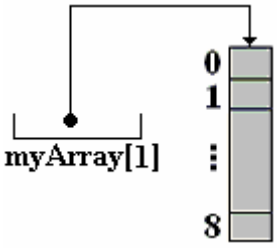
int [ ][ ] myArray = **new int** [n][ ];

définissent un sous-tableau de 10 pointeurs qui vont chacun pointer vers un tableau unidimensionnel qu'il faut instancier :

myArray[0] = **new int** [7];		instancie un tableau d'**int** unidimensionnel à 7 cases et renvoie sa référence qui est rangée dans la cellule de rang 0 du sous-tableau.
myArray[1] = **new int** [9];		instancie un tableau d'**int** unidimensionnel à 9 cases et renvoie sa référence qui est rangée dans la cellule de rang 1 du sous-tableau. Etc….

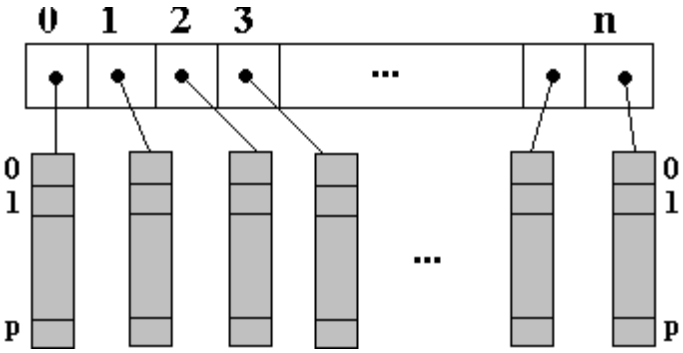
Attention

Dans le cas d'un tableau déchiqueté, le champ **Length** de la classe **Array**, contient **la taill**e du sous-tableau uni-dimensionnel associé à la référence.

Soit la déclaration : **int** [ ][ ] myArray = **new int** [n][ ]; **myArray** est une référence vers un sous-tableau de pointeurs. **myArray**.Length vaut 10 (taille du sous-tableau pointé)	
Soit la déclaration : myArray[0] = **new int** [7]; **MyArray** [0] est une référence vers un sous-tableau de cellules d'éléments de type **int**. **myArray**[0].Length vaut 7 (taille du sous-tableau pointé)	
Soit la déclaration : myArray[1] = **new int** [9]; **MyArray** [1] est une référence vers un sous-tableau de cellules d'éléments de type **int**. **myArray**[1].Length vaut 9 (taille du sous-tableau pointé)	

C# initialise les tableaux par défaut à **0** pour les **int**, **byte**, ... et à **null** pour les objets.

On peut simuler une matrice avec un tableau déchiqueté dont tous les sous-tableaux ont exactement la même dimension. Voici une figuration d'une matrice à n+1 lignes et à p+1 colonnes avec un tableau en escalier :

- Contrairement à Java qui l'accepte, le code ci-dessous ne sera pas compilé par C# : **int** [ ][ ] table = **new int** [n+1][p+1]; **- Il est nécessaire de créer manuellement tous les sous-tableaux :** **int** [ ][ ] table = **new int** [n+1][ ]; **for** (**int** i=0; i<n+1; i++) table[i] = **new int** [p+1];	

Conseil

L'exemple précédent montre à l'évidence que si l'on souhaite réellement utiliser des matrices en C#, il est plus simple d'utiliser la notion de tableau multi-dimensionnel **[,]** que celle de tableau en escalier **[ ] [ ].**

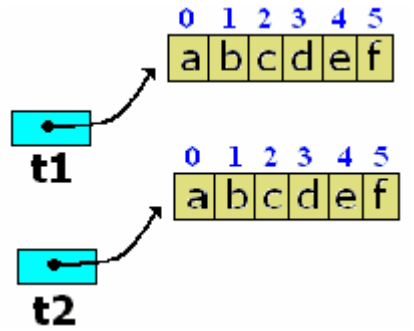
Egalité et inégalité de tableaux

L'opérateur d'égalité = = appliqué au tableau de n'importe quel type, détermine si deux objets spécifiés ont la même référence, il se comporte dans ce cas comme la méthode **Equals** de la classe Object qui ne teste que l'égalité de référence

int [ ] a , b ;
(a = = b) renvoie **true** si la référence a est la même que la référence b ; sinon il renvoie **false**.

Le morceau de code ci-dessous créé deux tableaux de char t1 et t2, puis teste leur égalité avec l'opérateur = = et la méthode Equals :

```
char[ ] t1="abcdef".ToCharArray();
char[ ] t2="abcdef".ToCharArray();
```




```
if(t1==t2)System.Console.WriteLine("t1=t2");
  else System.Console.WriteLine("t1<>t2");
if(t1.Equals(t2))System.Console.WriteLine("t1 égal t2");
  else System.Console.WriteLine("t1 différent de t2");
```

Après exécution on obtient :

```
t1<>t2
t1 différent de t2
```

Ces deux objets (les tableaux) sont différents (leurs références pointent vers des blocs différents)bien que le contenu de chaque objet soit le même.

Recopie de tableaux quelconques

Comme les tableaux sont des objets, l'affectation de références de deux tableaux distincts donne les mêmes résultats que pour d'autres objets : les deux références de tableaux pointent vers le même objet. Donc une affectation d'un tableau dans un autre t1 = t2 ne provoque pas la recopie des éléments du tableau t2 dans celui de t1.

Si l'on souhaite que t1 soit une copie identique de t2, tout en conservant le tableau t2 et sa référence distincte il faut utiliser l'une des deux méthodes suivante de la classe abstraite mère Array :

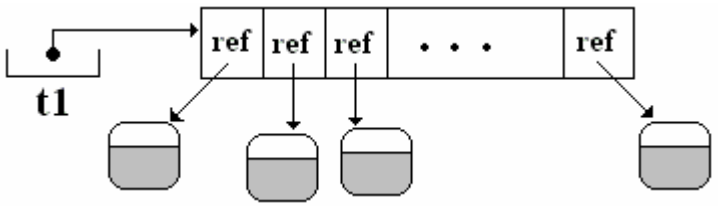
public virtual object **Clone( )** : méthode qui renvoie une référence sur une nouvelle instance de

tableau contenant les mêmes éléments que l'objet de tableau qui l'invoque. (il ne reste plus qu'à transtyper la référence retournée puisque clone renvoie un type object)

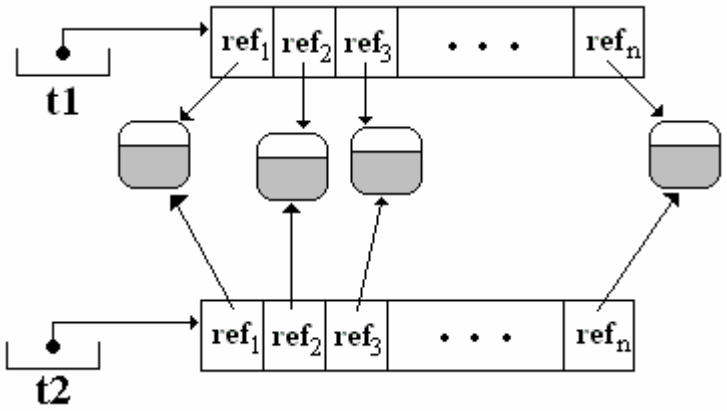
public static void **Copy** (**Array** t1 , **Array** t2, **int** long) : méthode de classe qui copie dans un tableau t2 déjà existant et déjà instancié, long éléments du tableau t1 depuis son premier élément (si l'on veut une copie complète du tableau t1 dans t2, il suffit que long représente le nombre total d'éléments soit long = t1.Length).

Attention

Dans le cas où le tableau t1 contient des références qui pointent vers des objets :



la recopie dans un autre tableau à travers les méthode Clone ou Copy ne recopie que les références, mais pas les objets pointés, voici un "clone" du tableau t1 de la figure précédente dans le tableau t2 :



Si l'on veut que le clonage (la recopie) soit plus complète et comprenne aussi les objets pointés, il suffit de construire une telle méthode car malheureusement la classe abstraite **Array** n'est pas implantable par l'utilisateur mais seulement par le compilateur et nous ne pouvons pas redéfinir la méthode virtuelle Clone).

Code source d'utilisation de ces deux méthodes sur un tableau unidimensionnel et sur une matrice :

```
//-- tableau à une dimension :

int[ ] t = new int[10];
for (int i=0; i<10; i++)
      t[i]=10*i;

int[ ] tab;
tab = ( int[ ] )t.Clone( );

int[ ] table = new int[10];
Array.Copy ( t , table , t.Length );
```

Déclaration avec définition explicite de taille.

Déclaration de référence, puis clonage et transtypage.

Déclaration avec définition explicite nécessaire de la taille, afin de pouvoir lancer la copie.

```
//-- tableau à deux dimensions :

int [ , ] matr = new int [3 , 2];
for (int i = 0; i<3; i++)
   for (int j = 0; j<2; j++)
         matr [i , j] = i+j;

int [ , ] matrtab;
matrtab=( int [ , ] )matr.Clone( );

int [ , ] matrtable = new int[3,2];
Array.Copy ( matr , matrtable , matr.Length );
```

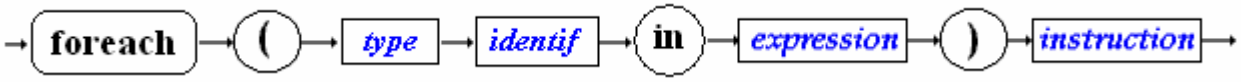
Déclaration de référence, puis clonage et transtypage.

Déclaration avec définition explicite nécessaire de la taille, afin de pouvoir lancer la copie.

Parcours itératifs de tableaux - foreach…in

Les instructions itératives for(…), while, do…while précédemment vues permettent le parcours d'un tableau élément par élément à travers l'indice de tableau. Il existe une instruction d'itération spécifique **foreach…in** qui énumère les éléments d'une collection, en exécutant un ensemble d'actions pour chaque élément de la collection.

Syntaxe



La classe Array est en fait un type de collection car elle implémente l'interface ICollection :

public abstract class Array : ICloneable, IList, ICollection, IEnumerable

Donc tout objet de cette classe (un tableau) est susceptible d'être parcouru par un instruction **foreach…in**. Mais les éléments ainsi parcourus ne peuvent être utilisés qu'en lecture, ils ne peuvent pas être modifiés, ce qui limite d'une façon importante la portée de l'utilisation d'un **foreach…in**.

foreach…in dans un tableau uni-dimensionnel

Dans un tableau **T** à une dimension de taille **long**, les éléments sont parcourus dans l'ordre croissant de l'index en commençant par la borne inférieure 0 et en terminant par la borne supérieure **long**-1 (rappel : **long** = **T**.Length).

Dans l'exemple ci-après où un tableau uni-dimensionnel table est instancié et rempli il y a équivalence de parcours du tableau table entre l'instruction **for** de gauche et l'instruction **foreach** de droite :

int [ ] table = **new int** [10]; …. Remplissage du tableau	
for (**int** i=0; i<10; i++) System.Console.WriteLine (table[i]);	**foreach** (**int** val **in** table) System.Console.WriteLine (val);

foreach…in dans un tableau multi-dimensionnel

Lorsque **T** est un tableau multi-dimensionnel microsoft indique : … les éléments sont parcourus de manière que les indices de la dimension la plus à droite soient augmentés en premier, suivis de ceux de la dimension immédiatement à gauche, et ainsi de suite en continuant vers la gauche.

Dans l'exemple ci-après où une matrice table est instanciée et remplie il y a équivalence de parcours de la matrice table, entre l'instruction **for** de gauche et l'instruction **foreach** de droite (*fonctionnement identique pour les autres types de tableaux multi-dimensionnels et en escalier*) :

int [,] table = **new int** [3 , 2]; …. Remplissage de la matrice	
for (**int** i=0; i<3; i++) **for** (**int** j=0; j<2; i++) System.Console.WriteLine (table[i , j]);	**foreach** (**int** val **in** table) System.Console.WriteLine (val);

Avantage : la simplicité d'écriture, toujours la même quelle que soit le type du tableau.

Inconvénient : on ne peut qu'énumérer en lecture les éléments d'un tableau.

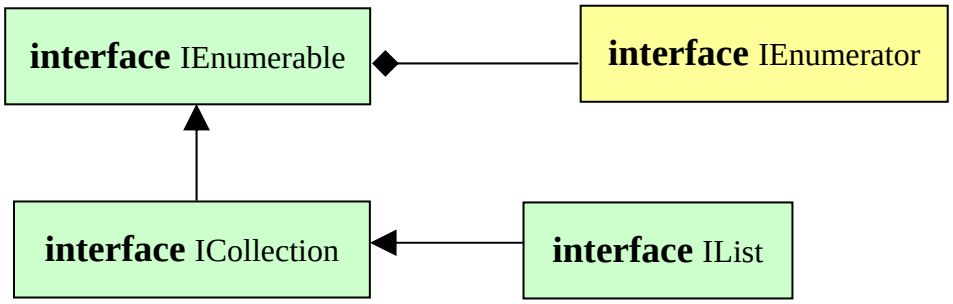
Collections - Piles - Files - Listes

Généralités sur les collections

Ce chapitre est à aborder si le lecteur est familiarisé avec les concepts objets, dans le cas contraire le lecteur sautera ce chapitre pour y revenir ultérieurement.

L'espace de noms **System.Collections** contient des interfaces et des classes qui permettent de manipuler des collections d'objets. Plus précisément, les données structurées classiques que l'on utilise en informatique comme les listes, les piles, les files d'attente,… sont représentées dans .Net framework par des classes directement utilisables du namespace **System.Collections**.

Quatre interfaces de cet espace de noms jouent un rôle fondamental : IEnumerable, IEnumerator, ICollection et IList selon les diagrammes d'héritage et d'agrégation suivants :



IEnumerable :

contient une seule méthode qui renvoie un énumérateur (objet de type IEnumerator) qui peut itérer sur les élément d'une collection (c'est une sorte de pointeur qui avance dans la collection, comme un pointeur de fichier se déplace sur les enregistrements du fichier) :

public IEnumerator GetEnumerator( );

IEnumerator :

Propriétés

public object Current {get;} Obtient l'élément en cours pointé actuellement par l'énumérateur dans la collection.

Méthodes

public bool MoveNext( ); Déplace l'énumérateur d'un élément il pointe maintenant vers l'élément suivant dans la collection (renvoie **false** si l'énumérateur est après le dernier élément de la collection sinon renvoie **true**).

public void Reset( ); Déplace l'énumérateur au début de la collection, **avant** le premier élément (donc si l'on effectue un Current on obtiendra la valeur **null**, car après un Reset( ), l'énumérateur ne pointe pas devant le premier élément de la collection mais avant ce premier élément !).

ICollection :

Propriétés

public int Count {get;}	Fournit le nombre d'éléments contenus dans **ICollection**.
public bool IsSynchronized {get;}	Fournit un booléen indiquant si l'accès à **ICollection** est synchronisé (les éléments de **ICollection** sont protégés de l'accès simultanés de plusieurs threads différents).
public object SyncRoot {get;}	Fournit un objet qui peut être utilisé pour synchroniser (verrouiller ou déverrouiller) l'accès à **ICollection**.

Méthode

public void CopyTo (Array table, **int** index**)**	Copie les éléments de **ICollection** dans un objet de type Array (table), commençant à un index fixé.

IList :

Propriétés

public bool IsFixedSize {get;} : indique si **IList** est de taille fixe.

public bool IsReadOnly {get;} : indique si **IList** est en lecture seule.

Les classes implémentant l'interface IList sont indexables par l'indexeur [ ].

Méthodes (classique de gestion de liste)

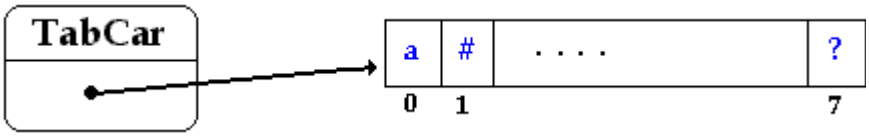
public int Add(object *elt* **);**	Ajoute l'élément *elt* à **IList**.
public void Clear( );	Supprime tous les éléments de **IList**.
public bool Contains(object *elt* **);**	Indique si **IList** contient l'élément *elt* en son sein.
public int IndexOf(object *elt* **);**	Indique le rang de l'élément *elt* dans **IList**.
public void Insert(int *rang* **,** object *elt* **);**	Insère l'élément *elt* dans **IList** à la position spécifiée par *rang*.
public void Remove(object *elt* **);**	Supprime la première occurrence de l'objet *elt* de **IList**.
public void RemoveAt (int *rang***);**	Supprime l'élément de **IList** dont le *rang* est spécifié.

Ces quatre interfaces C# servent de contrat d'implémentation à de nombreuses classes de structures de données, nous en étudions quelques unes sur le plan pratique dans la suite du document.

Les tableaux dynamiques : classe ArrayList

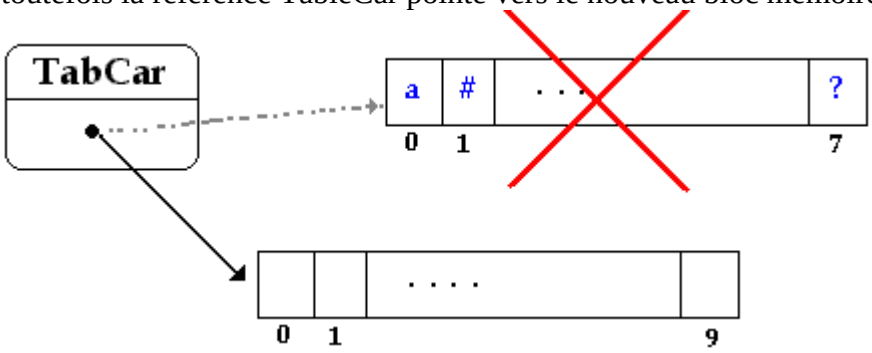
Un tabeau Array à une dimension, lorsque sa taille a été fixée soit par une définition explicite, soit par une définition implicite, **ne peut plus changer de taille**, c'est donc un objet de taille statique.

char [ ] TableCar ;
TableCar = **new char**[8]; *//définition de la taille et création d'un nouvel objet tableau à 8 cellules*
TableCar[0] = 'a';
TableCar[1] = '#';
...
TableCar[7] = '?';



Si l'on rajoute l'instruction suivante aux précédentes

< TableCar = **new char**[10]; > il y a création d'un nouveau tableau de même nom et de taille 10, l'ancien tableau à 8 cellules est alors détruit. Nous ne redimensionnons pas le tableau, mais en fait nous créons un nouvel objet utilisant la même variable de référence TableCar que le précédent, toutefois la référence TableCar pointe vers le nouveau bloc mémoire :



Ce qui nous donne après exécution de la liste des instructions ci-dessous, un tableau TabCar ne contenant plus rien :
char [ ] TableCar ;
TableCar = **new char**[8];
TableCar[0] = 'a';
TableCar[1] = '#';
...
TableCar[7] = '?';
TableCar = **new char**[10];

Comment faire pour "agrandir" un tableau pendant l'exécution

- Il faut déclarer un nouveau tableau t2 plus grand,
- puis recopier l'ancien dans le nouveau, par exemple en utilisant la méthode **public static** void **Copy** (**Array** t1 , **Array** t2, **int** long)

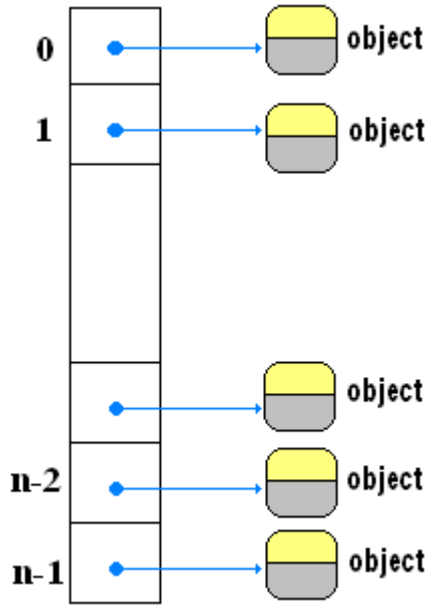
Il est possible d'éviter cette façon de faire en utilisant une classe de vecteur (tableau unidimensionnel dynamique) qui est en fait une liste dynamique gérée comme un tableau.

La classe concernée se dénomme System.Collections.ArrayList, elle hérite de la classe object et implémente les interfaces IList, ICollection, IEnumerable, ICloneable
(public class ArrayList : IList, ICollection, IEnumerable, ICloneable;**)**

Les principales méthodes permettant de manipuler les éléments d'un* ArrayList *sont :

public virtual int Add(object value);	Ajoute l'objet value à la fin de ArrayList.
public virtual void Insert(int index, object value);	Insère un élément dans ArrayList à l'index spécifié.
public virtual void Clear();	Supprime tous les éléments de ArrayList.
public virtual void Remove(object obj);	Supprime la première occurrence d'un objet spécifique de ArrayList.
public virtual void Sort( );	Trie les éléments dans l'intégralité de ArrayList à l'aide de l'implémentation IComparable de chaque élément (algorithme QuickSort).
ArrayList Table; Table[i] = ...;	Accès en lecture et en écriture à un élément quelconque de rang i du tableau par Table[i]
PROPRIETE	
public virtual int Count { **get** ;}	Vaut le nombre d'éléments contenus dans ArrayList, **propriété** en lecture seulement..
[ ]	Propriété indexeur de la classe, on l'utilise comme un opérateur tab**[** i **]** accède à l'élément de rang i.

Schéma interne d'un ArrayList :



Où **n** est le nombre d'éléments présent dans l'ArrayList
(n = this.Count).

L'accès à un élément d'un objet de type ArrayList s'effectue essentiellement grâce à l'indexeur (R/W) :

```
ArrayList liste ;
…
object obj;
….
int k = liste.Add( obj );
…
object x = liste[k] ;
…
liste[k] = new object() ;
```

Un objet de classe **ArrayList** peut "grandir" automatiquement d'un certain nombre de cellules pendant l'exécution, c'est le programmeur qui peut fixer la valeur d'augmentation du nombre de cellules supplémentaires dès que la capacité maximale en cours est dépassée. Dans le cas où la valeur d'augmentation n'est pas fixée, c'est la machine virtuelle du CLR qui procède à une augmentation par défaut.

Vous pouvez utiliser le type **ArrayList** avec n'importe quel type d'objet puisqu'un **ArrayList** contient des éléments de type dérivés d'object (ils peuvent être tous de types différents et le vecteur est de type hétérogène).

Voici un exemple simple de vecteur de chaînes utilisant quelques unes des méthodes précédentes :

```
static void afficheVector (ArrayList vect)  //affiche un vecteur de string
 {
   System.Console.WriteLine( "Vecteur taille = " + vect.Count );
   for ( int i = 0; i<= vect.Count-1; i++ )
       System.Console.WriteLine( "Vecteur[" + i + "]=" + (string)vect[ i ] );
 }

static void VectorInitialiser ( ) // initialisation du vecteur de string
{    ArrayList table = new ArrayList( );
     string str = "val:";
     for ( int i = 0; i<=5; i++ )
          table.Add(str + i.ToString( ) );
     afficheVector(table);
}
```

*Voici le résultat de l'exécution de la méthode***VectorInitialiser** ***:***

```
Vecteur taille = 6
Vecteur [0] = val:0
Vecteur [1] = val:1
Vecteur [2] = val:2
Vecteur [3] = val:3
Vecteur [4] = val:4
Vecteur [5] = val:5
```

IList
ICollection
IEnumerable
ICloneable

ArrayList
Class

Classe **ArrayList**

Properties

- Capacity { get; set; } : int
- Count { get; } : int
- IsFixedSize { get; } : bool
- IsReadOnly { get; } : bool
- IsSynchronized { get; } : bool
- Item[int index] { get; set; } : object
- SyncRoot { get; } : object

Methods

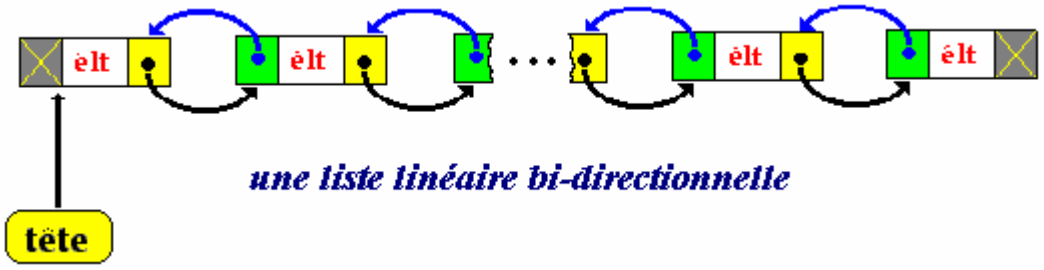
- Adapter(IList list) : ArrayList
- Add(object value) : int
- AddRange(ICollection c) : void
- ArrayList()
- ArrayList(ICollection c)
- ArrayList(int capacity)
- BinarySearch(int index, int count, object value, IComparer comparer) : int
- BinarySearch(object value) : int
- BinarySearch(object value, IComparer comparer) : int
- Clear() : void
- Clone() : object
- Contains(object item) : bool
- CopyTo(Array array) : void
- CopyTo(Array array, int arrayIndex) : void
- CopyTo(int index, Array array, int arrayIndex, int count) : void
- FixedSize(ArrayList list) : ArrayList
- FixedSize(IList list) : IList
- GetEnumerator() : IEnumerator
- GetEnumerator(int index, int count) : IEnumerator
- GetRange(int index, int count) : ArrayList
- IndexOf(object value) : int
- IndexOf(object value, int startIndex) : int
- IndexOf(object value, int startIndex, int count) : int
- Insert(int index, object value) : void
- InsertRange(int index, ICollection c) : void
- LastIndexOf(object value) : int
- LastIndexOf(object value, int startIndex) : int
- LastIndexOf(object value, int startIndex, int count) : int
- ReadOnly(ArrayList list) : ArrayList
- ReadOnly(IList list) : IList
- Remove(object obj) : void
- RemoveAt(int index) : void
- RemoveRange(int index, int count) : void
- Repeat(object value, int count) : ArrayList
- Reverse() : void
- Reverse(int index, int count) : void
- SetRange(int index, ICollection c) : void
- Sort() : void
- Sort(IComparer comparer) : void
- Sort(int index, int count, IComparer comparer) : void
- Synchronized(ArrayList list) : ArrayList
- Synchronized(IList list) : IList
- ToArray() : object[]
- ToArray(Type type) : Array
- TrimToSize() : void

Les listes chaînées : classe ArrayList

Rappelons qu'une liste linéaire (ou liste chaînée) est un ensemble ordonné d'éléments de même type (structure de donnée homogène) auxquels on accède séquentiellement. Les opérations minimales effectuées sur une liste chaînée sont l'insertion, la modification et la suppression d'un élément quelconque de la liste.

Les listes peuvent être uni-directionnelles, elles sont alors parcourues séquentiellement dans un seul sens :

ou bien bi-directionnelles dans lesquelles chaque élément possède deux liens de chaînage, l'un sur l'élément qui le suit, l'autre sur l'élément qui le précède, le parcours s'effectuant en suivant l'un ou l'autre sens de chaînage :



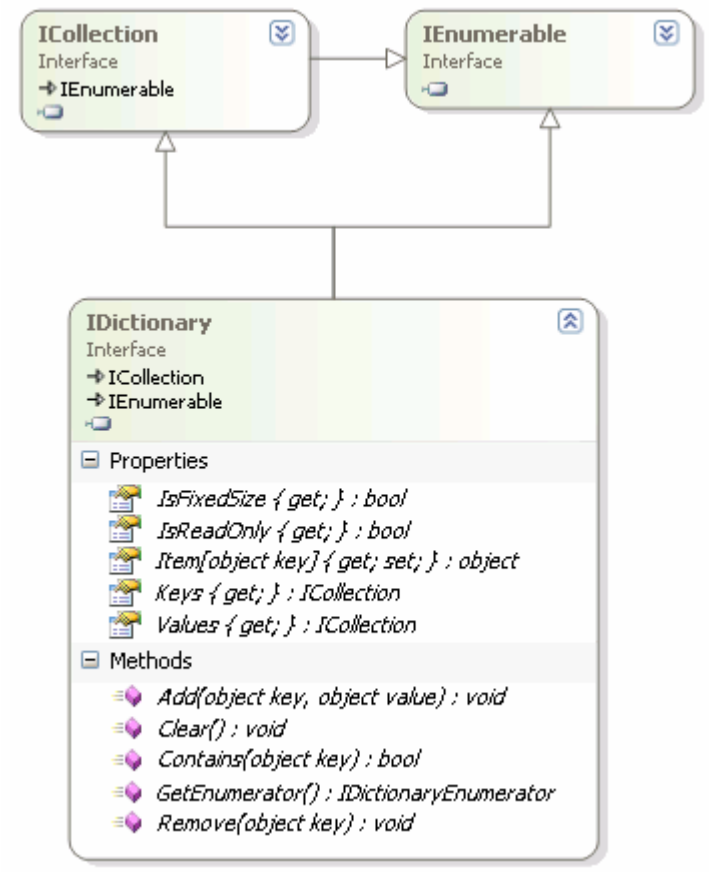
La classe **ArrayList** peut servir à une implémentation de la liste chaînée uni ou bi-directionnelle; un **ArrayList** contient des éléments de type dérivés d'Object, la liste peut donc être hétérogène, cf exercice sur les listes chaînées.

Liste à clefs triées : classe SortedList

Si l'on souhaite gérer une liste triée par clef, il est possible d'utiliser la classe **SortedList** (localisation : **System.Collections.SortedList**). Cette classe représente une collection de paires valeur-clé triées par les clés toutes différentes et accessibles par clé et par index : il s'agit donc d'une liste d'identifiants et de valeur associée à cet identifiant, par exemple une liste de personne dont l'identifiant (la clef) est un entier et la valeur associée des informations sur cette personne sont stockées dans une structure (le nom, l'âge, le genre, ...).

Les interfaces implémentées par SortedList

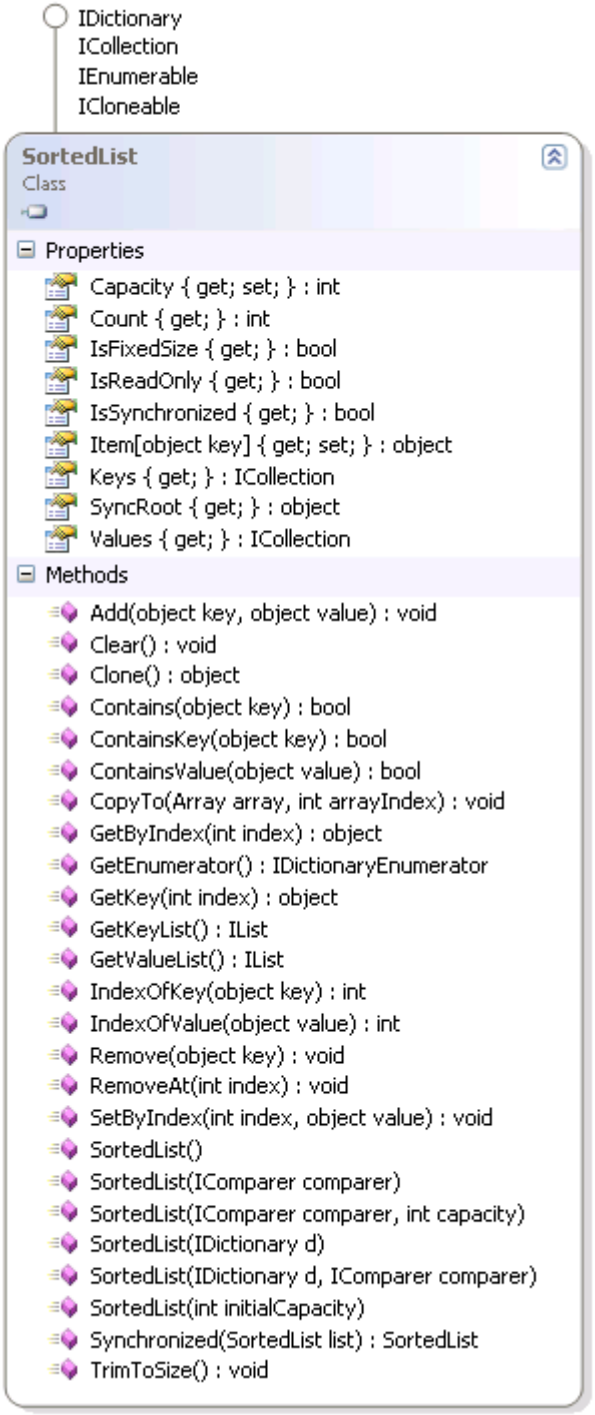
Interface	Description
ICollection	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
ICloneable	Contrat d'une seuleméthode pour cloner (recopier entièrement) un objet : **object** clone( )
IEnumerable	Contrat pour l'énumération d'une collection.
IDictionary	Représente une collection sous forme de couple (clef, valeur).



Cette classe n'est pas utile pour la gestion d'une liste chaînée classique non rangée à cause de son tri automatique selon les clefs.

En revanche, si l'on stocke comme clef la valeur de Hashcode de l'élément, la recherche est améliorée.

La classe SortedList :



Structure utilisée par SortedList

Structure	Description
DictionaryEntry	Défini un unique couple (clef, valeur) qui peut être stocké ou retrouvé. **public struct** DictionaryEntry

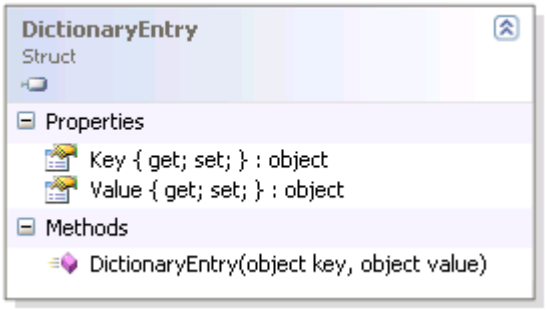
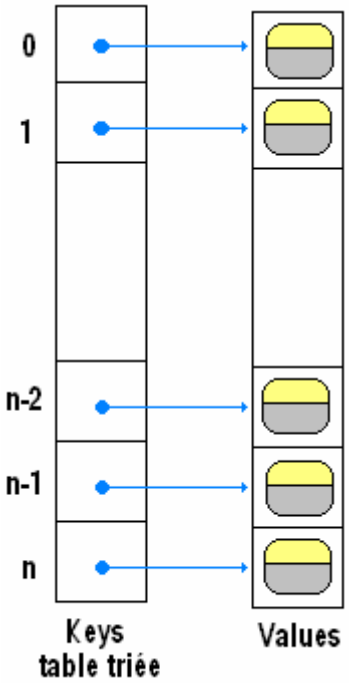


Schéma interne d'un SortedList :

(un SortedList = 2 tables **Keys** et **Values** liées entre elles implantant un IDictionnary)

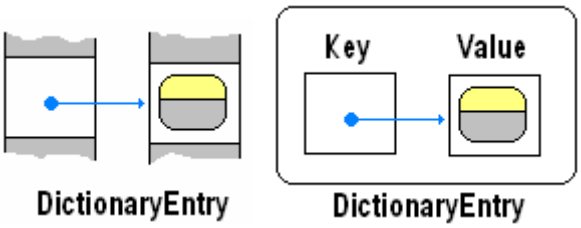


Où **n** est le nombre d'éléments présent dans le **SortedList** (n = this.Count).

1°) Accès à un élément essentiellement grâce à sa clef et à l'indexeur associé :

```
SortedList liste ;
….
liste.Add( objKey, objValue) ;
....
object x = liste[objKey ] ;
....
liste[objKey ] = new object( ) ;
```

2°) Le couple (une cellule de **Keys** , la cellule associée dans **Values**) peut aussi être considéré comme un **DictionaryEntry** = (**Key** ,**Value**), le **SortedList** peut alors être considéré aussi comme une collection d'objets de type **DictionaryEntry** :



Dans ce cas l'accès aux éléments d'un **SortedList** peut aussi s'effectuer avec une boucle **foreach** sur chaque **DictionaryEntry** contenu dans le **SortedList**.

```
foreach (DictionaryEntry couple in this ) {
   couple.Key.... (accès à la clef)
   couple.Value....(accès à la valeur associée à la clef)
}
```

Les principales méthodes permettant de manipuler les éléments d'un SortedList *sont :*

public virtual void Add(object key, object value);	Ajoute un élément avec la clé key et la valeur value spécifiées dans le **SortedList**.
public virtual void CopyTo(**Array** array, **int** arrayIndex);	Copie les éléments du **SortedList** dans une instance **Array** array unidimensionnelle à l'index arrayIndex spécifié (valeur de l'index dans array où la copie

	commence).
public virtual void Clear( );	Supprime tous les éléments de **SortedList**.
public virtual object GetByIndex(**int** index);	Obtient la valeur à l'index spécifié de la liste **SortedList**.
public virtual object GetKey(**int** index);	Obtient la clé à l'index spécifié de **SortedList**.
public virtual int IndexOfValue(object value);	Retourne l'index de base zéro de la première occurrence de la valeur value spécifiée dans **SortedList**.
public virtual int IndexOfKey(object key);	Retourne l'index de base zéro de la clé key spécifiée dans **SortedList**.
public virtual void Remove(object key);	Supprime de **SortedList** l'élément ayant la clé key spécifiée.
public virtual void RemoveAt(**int** index);	Supprime l'élément au niveau de l'index spécifié de **SortedList**.
public virtual IList GetValueList ( );	Obtient un objet de liste IList en lecture seule contenant toutes les valeurs triées dans le même ordre que dans le **SortedList.**
PROPRIETE	
public virtual int Count { **get** ; }	Vaut le nombre d'éléments contenus dans **SortedList**, **propriété** en lecture seulement.
[ ]	Propriété indexeur de la classe, on l'utilise comme un opérateur tab**[** i **]** accède à l'élément dont la clef vaut i.
public virtual ICollection Values { **get**; }	Obtient dans un objet de ICollection les valeurs dans **SortedList**. (les éléments de ICollection sont tous triés dans le même ordre que les valeurs du **SortedList**)
public virtual ICollection Keys { **get**; }	Obtient dans un objet de ICollection les clés dans **SortedList**. (les éléments de ICollection sont tous triés dans le même ordre que les clefs du **SortedList**)

Exemple d'utilisation d'un SortedList :

```
SortedList Liste = new SortedList ( );
Liste.Add(100,"Jean");
Liste.Add(45,"Murielle");
Liste.Add(201,"Claudie");
Liste.Add(35,"José");
Liste.Add(28,"Luc");

 //----> Balayage complet de la Liste par index :
for (int i=0; i<Liste.Count; i++)
  System.Console.WriteLine( (string)Liste.GetByIndex(i) );

//----> Balayage complet de la collection des valeurs :
foreach(string s in Liste.Values)
  System.Console.WriteLine( s );
```

```
//----> Balayage complet de la collection des clefs :
foreach(object k in Liste.Keys)
  System.Console.WriteLine( Liste[k] );

//----> Balayage complet de l'objet IList retourné :
for (int i = 0; i < Liste.GetValueList( ).Count; i++)
   System.Console.WriteLine( Liste.GetValueList( ) [ i ] );
```

Soit la représentation suivante (attention à la confusion entre clef et index) :

28,"Luc"	35,"José"	45,"Murielle"	100,"Jean"	201,"Claudie"
0	1	2	3	4

Liste.GetByIndex(2) : Murielle

Liste [45] : Murielle

Liste.GetValueList( ) [2] : Murielle

Les trois boucles affichent dans l'ordre :

Luc
José
Murielle
Jean
Claudie

Piles Lifo, files Fifo : classes Stack et Queue

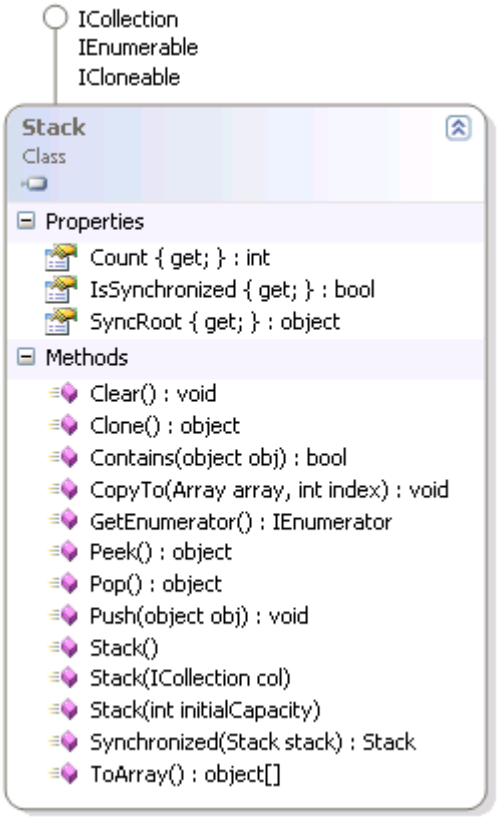
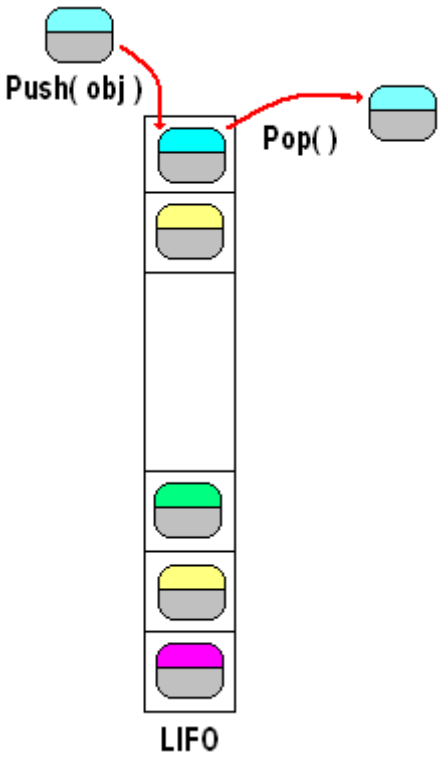
Ces deux classes font partie du namespace **System.Collections :**

System.Collections. Stack

System.Collections. Queue

Les interfaces implémentées par Stack

Interface	Description
ICollection	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
ICloneable	Contrat d'une seuleméthode pour cloner (recopier entièrement) un objet : **object** clone( )
IEnumerable	Contrat pour l'énumération d'une collection.

La classe Stack :	Schéma interne d'un objet LIFO de classe Stack :
	

La classe "**public class** Stack : ICollection, IEnumerable, ICloneable" représente une pile Lifo :

public virtual object **Peek ( );**	Renvoie la référence de l'objet situé au sommet de la pile.
public virtual object **Pop( );**	Dépile la pile (l'objet au sommet est enlevé et renvoyé)
public virtual void **Push(** object elt **);**	Empile un objet au sommet de la pile.
public virtual object **[ ] ToArray( );**	Recopie toute la pile dans un tableau d'objet depuis le sommet jusqu'au fond de la pile (dans l'ordre du dépilement).

Les interfaces implémentées par Queue

Interface	Description
ICollection	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
ICloneable	Contrat d'une seuleméthode pour cloner (recopier entièrement) un objet : **object** clone( )
IEnumerable	Contrat pour l'énumération d'une collection.

La classe Queue :

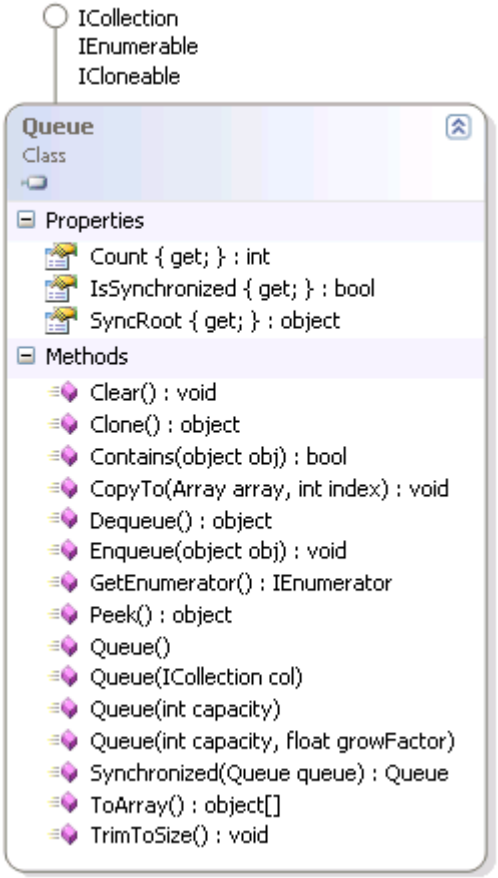
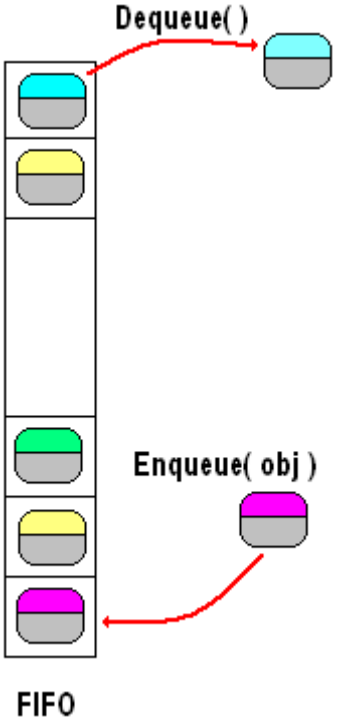


Schéma interne d'un objet FIFO de classe Queue :



La classe "**public class** Queue : ICollection, IEnumerable, ICloneable" représente une file Fifo :

public virtual object **Peek ( );**	Renvoie la référence de l'objet situé au sommet de la file.
public virtual object **Dequeue( );**	L'objet au début de la file est enlevé et renvoyé.
public virtual void **Enqueue (** object elt **);**	Ajoute un objet à la fin de la file.
public virtual object **[ ] ToArray( );**	Recopie toute la file dans un tableau d'objet depuis le début de la fifo jusqu'à la fin de la file.

Exemple d'utilisation d'une Lifo de type Stack

Construisons une pile de **string** possédant une méthode getArray permettant d'empiler immédiatement dans la pile tout un tableau de **string**.

Le programme ci-dessous rempli avec les chaînes du tableau t1 grâce à la méthode getArray, la pile Lifo construite. On tente ensuite de récupérer le contenu de la pile sous forme d'un tableau de chaîne t2 (opération inverse) en utilisant la méthode ToArray. Le compilateur signale une erreur :

```
class Lifo : Stack
{
   public virtual void getArray(string[ ] t) {
      foreach(string s in t)
            this.Push (s);
   }
}
```

```
class Class
{
   static void Main ( string[ ] args )          {
      Lifo piLifo = new Lifo ( );
      string [ ] t1 =  {"aaa","bbb","ccc","ddd","eee","fff","fin"};
      string [ ] t2 ;
      piLifo.getArray(t1) ;
      t2 = piLifo.ToArray( ) ;
      foreach ( string s  in  t2 )
         System.Console.WriteLine(s) ;
}
```

Impossible de convertir implicitement le type 'object[ ]' en 'string[ ]' .

En effet la méthode ToArray renvoie un tableau d'object et non un tableau de string. On pourrait penser à transtyper explicitement :

```
t2 = ( string [ ] ) piLifo.ToArray( ) ;
```

en ce cas C# réagit comme Java, en acceptant la compilation, mais en générant une exception de cast invalide, car il est en effet dangereux d'accepter le transtypage d'un tableau d'object en un tableau de quoique ce soit, car chaque object du tableau peut être d'un type quelconque et tous les types peuvent être différents !

Il nous faut donc construire une méthode ToArray qui effectue le transtypage de chaque cellule du tableau d'object et renvoie un tableau de string, or nous savons que la méthode de classe Array nommée Copy un tableau t1 vers un autre tableau t2 en effectuant éventuellement le transtypage des cellules : Array.Copy(t1 , t2 , t1.Length)

Voici le code de la nouvelle méthode ToArray :

```
class Lifo : Stack
{
   public virtual void getArray ( string[ ] t ) {
      foreach(string s in t)
            this.Push (s);
   }
   public new virtual string [ ] ToArray ( ){
      string[ ] t = new string [this.Count];
      Array.Copy( base.ToArray( ), t , this.Count );
      return t ;
   }
}
```

```
class Class
{
   static void Main ( string[ ] args )          {
      Lifo piLifo = new Lifo ( );
      string [ ] t1 =  {"aaa","bbb","ccc","ddd","eee","fff","fin"};
      string [ ] t2 ;
      piLifo.getArray(t1) ;
      t2 = piLifo.ToArray( ) ;
      foreach ( string s  in  t2 )
         System.Console.WriteLine(s) ;
}
```

Appel à la méthode ToArray mère qui renvoie un object[ ]

fin
fff
eee
ddd
ccc
bbb
aaa

Nous avons mis le qualificateur **new** car cette méthode masque la méthode mère de la classe Stack, nous avons maintenant une pile Lifo de **string**.

Exemple d'utilisation d'une Fifo de type Queue

Construisons d'une manière identique à la construction précédente, une file de **string** possédant une méthode getArray permettant d'ajouter immédiatement dans la file tout un tableau de **string** et la méthode ToArray redéfinie :

Nous livrons immédiatement le code source de cette classe et celui de la classe d'appel :

```
class Fifo : Queue
{
   public virtual void getArray ( string[ ] t ) {
      foreach(string s in t)
            this. Enqueue (s);
   }
   public new virtual string [ ] ToArray ( ){
      string[ ] t = new string [this.Count];
      Array.Copy( base.ToArray( ), t , this.Count );
      return t ;
   }
}
```

```
class Class
{
   static void Main ( string[ ] args )          {
     string [ ] t1 =  {"aaa","bbb","ccc","ddd","eee","fff","fin"};
     string [ ] t2 ;
    Fifo filFifo = new Fifo ( );
    filFifo.getArray(t1);
    t2 = filFifo.ToArray( );
    foreach (string s in t2)
        System.Console.WriteLine(s);
    System.Console.ReadLine();
}
```

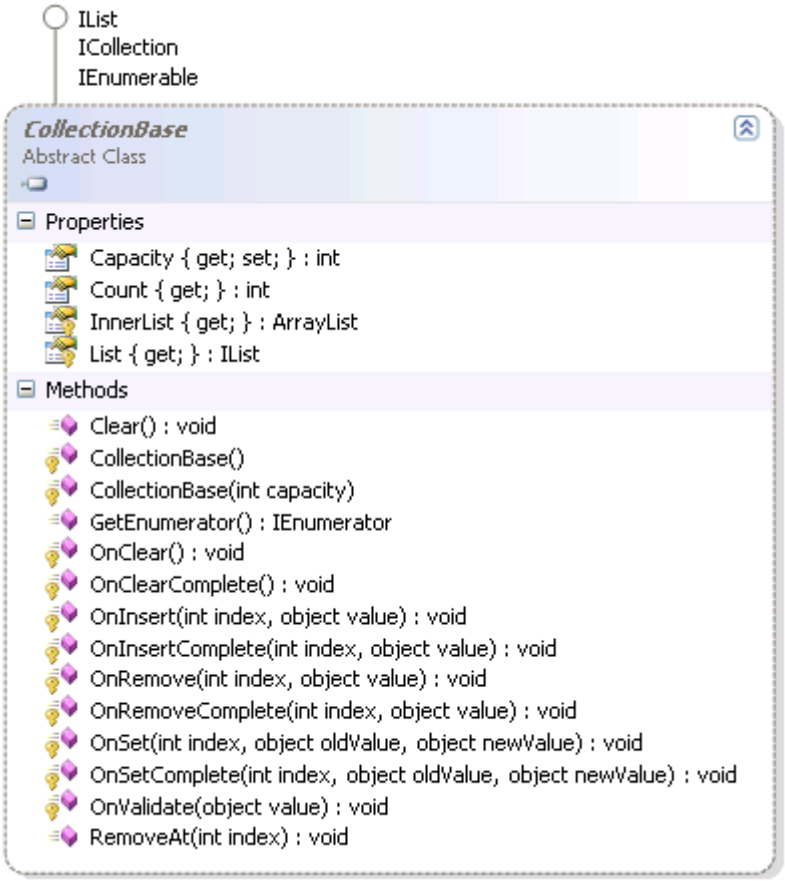
```
aaa
bbb
ccc
ddd
eee
fff
fin
```

Classes de bases pour collections personnalisées

CollectionBase

Il existe une classe abstraite de gestion d'une collection d'objets nommée CollectionBase, elle est située dans le namespace **System.Collections**.

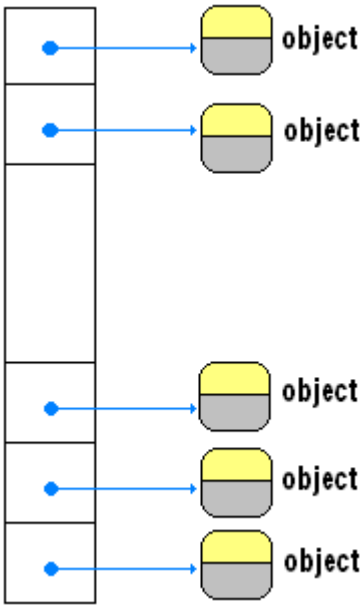
La classe CollectionBase :



Les interfaces implémentées par CollectionBase

Interface	Description
ICollection	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
IEnumerable	Contrat pour l'énumération d'une collection.
IList	Représente une collection dont chaque élément est accessible par un index.

Schéma interne d'un objet de classe CollectionBase :



Cette structure peut être atteinte soit comme un ArrayList, soit comme un IList.

Si l'on souhaite construire une collection personnalisée bénéficiant des fonctionnalités de base offertes par .Net, il faut hériter de la classe CollectionBase :

```
class MaCollection : CollectionBase { …  }
```

DictionaryBase

Il existe une classe abstraite de gestion d'une collection d'objets rangés sous forme de de dictionnaire (paire de valeur <clef,valeur>), nommée DictionaryBase, elle est située dans le namespace **System.Collections**.

Les interfaces implémentées par DictionaryBase

Interface	Description
ICollection	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
IEnumerable	Contrat pour l'énumération d'une collection.
IDictionary	Représente une collection sous forme de couple (clef, valeur).

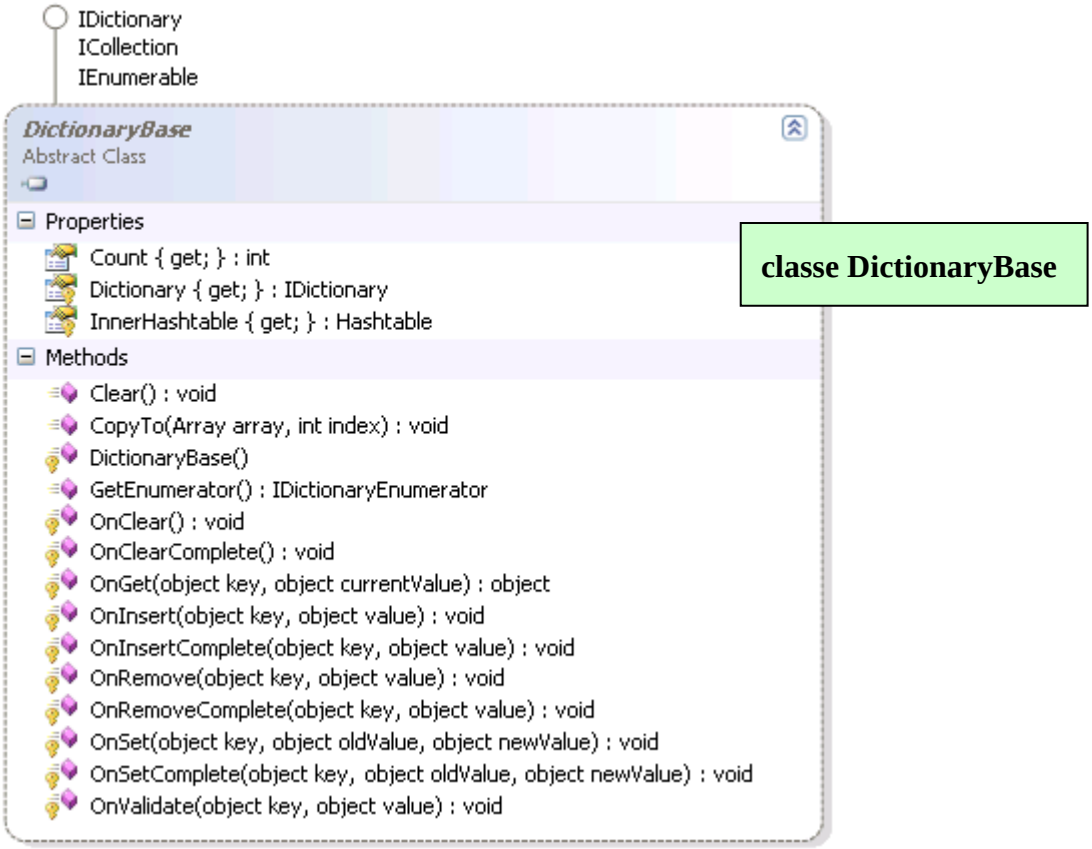
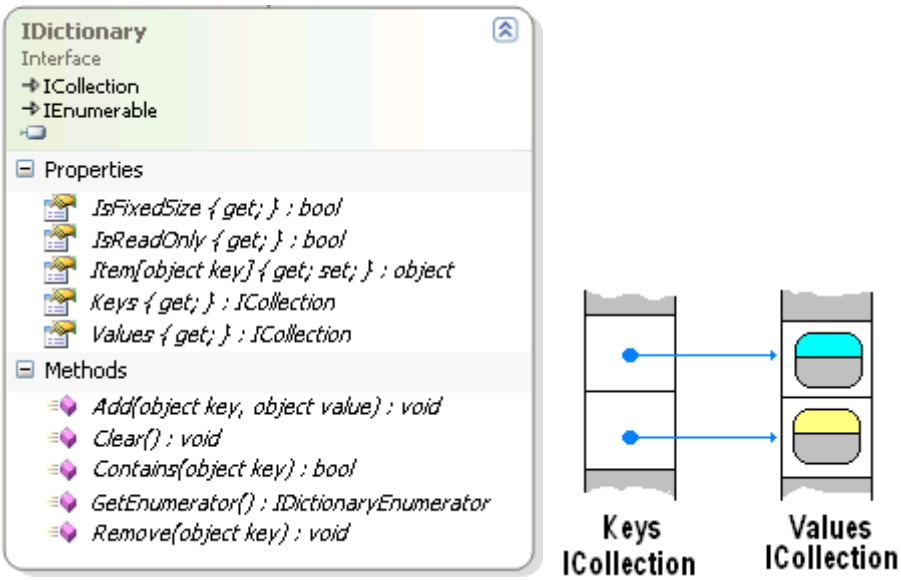


Schéma interne d'un objet de classe DictionaryBase :

Les données sont atteignables à travers la propriété Dictionary {get;}qui est un objet de type **IDictionary** :

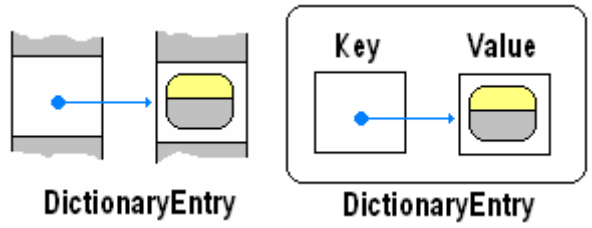


Si l'on souhaite construire une Dictionnaire personnalisé bénéficiant des fonctionnalités de base offertes par .Net, il faut hériter de la classe DictionaryBase.

class MonDictionnaire: DictionaryBase { … }

Accès aux données en lecture seulement directement :

Comme pour le SortedList, le couple (une cellule de **Keys** , la cellule associée dans **Values**) peut aussi être considéré comme un **DictionaryEntry** = (**Key** ,**Value**), un objet de classe MonDictionnaire peut alors être considéré aussi comme une collection d'objets de type **DictionaryEntry** :



Dans ce cas l'accès aux éléments d'un MonDictionnaire peut s'effectuer avec une boucle **foreach** sur chaque **DictionaryEntry** contenu dans le MonDictionnaire.

```
foreach (DictionaryEntry couple in this ) {
   couple.Key.... (accès à la clef)
   couple.Value....(accès à la valeur associée à la clef)
}
```

Accès aux données en lecture seulement à travers le Dictionary :

Comme cette classe abstraite met sa propriété Dictionary en accès protégé (**Protected ReadOnly Property** Dictionary As IDictionary), il est bon dans la classe fille construite d'augmenter le niveau de visibilité de cette propriété en la rendant par exemple publique afin que les données soient accessibles aux objets, on peut alors accéder aussi aux éléments d'un objet de classe MonDictionnaire à partir de sa propriété Dictionnary :

```
foreach (DictionaryEntry couple in this. Dictionary ){
   couple.Key.... (accès à la clef)
   couple.Value....(accès à la valeur associée à la clef)
}
```

La propriété Dictionary permet plus de manipulations sur les données puisqu'elle possède les méthodes Add, Contains, Remove et Clear qui agissent directement sur les données de l'objet de classe MonDictionnaire.

Exemple de classe dérivée de DictionaryBase :

```
class MonDictionnaire : DictionaryBase
{
   // on rend Public la propriété protected Dictionary :
   public new IDictionary Dictionary
   {
      get
      {
         return base.Dictionary;
```

```
        }
    }
    // Afichage des données à partir du Dictionary :
    public void afficherDico()
    {
        foreach (DictionaryEntry elt in this.Dictionary)
        {
            System.Console.WriteLine("   {0} : {1}", elt.Key, elt.Value);
        }
    }
    // Afichage des données à partir de l'objet lui-même :
    public void afficherMe()
    {
        foreach (DictionaryEntry elt in this)
        {
            System.Console.WriteLine(" {0}...{1}", elt.Key, elt.Value);
        }
    }
}

class Principale
{
    public static void Main(string[] args)
    {
        MonDictionnaire dico = new MonDictionnaire();
        dico.Dictionary.Add(10, "rmd1");
        dico.Dictionary.Add(30, "rmd2");
        dico.Dictionary.Add(20, "rmd3");
        dico.Dictionary.Add(50, "rmd4");
        dico.Dictionary.Add(40, "rmd5");
        dico.afficherDico();
        dico.afficherMe();
        Console.ReadLine();
    }
}
```

Résultats obtenus :

```
   10 : rmd1
   20 : rmd3
   30 : rmd2
   40 : rmd5
   50 : rmd4
 10...rmd1
 20...rmd3
 30...rmd2
 40...rmd5
 50...rmd4
```

Un langage très orienté objet

- Classes, objets et méthodes
- Polymorphisme d'objets
- Polymorphisme de méthodes
- Polymorphisme d'interfaces
- Classe de délégation
- Traitement d'exceptions
- Processus et multi-threading

Classes, objets et méthodes

Plan général:

1. Les classes C# : des nouveaux types

1.1 Déclaration d'une classe
1.2 Une classe est un type en C#
1.3 Toutes les classes ont le même ancêtre - héritage
1.4 Encapsulation des classes
1.5 Exemple de classe imbriquée dans une autre classe
1.6 Exemple de classe inclue dans un même espace de noms
1.7 Méthodes abstraites
1.8 Classe abstraite, Interface

2. Les objets : des références ou des valeurs

2.1 Modèle de la référence
2.2 Les constructeurs d'objets référence ou valeurs
2.3 Utilisation du constructeur d'objet par défaut
2.4 Utilisation d'un constructeur d'objet personnalisé
2.5 Le mot clef this- cas de la référence seulement

3. Variables et méthodes

3.1 Variables dans une classe en général
3.2 Variables et méthodes d'instance

3.3 Variables et méthodes de classe - static
3.4 Bilan et exemple d'utilisation

Introduction

Nous proposons des comparaisons entre les syntaxes de C# et Delphi et/ou Java, lorsque les définitions sont semblables.

Tableau des limitations des niveaux de visibilité fourni par microsoft :

Contexte	Notes
Classes	La classe de base directe d'un type de classe doit être au moins aussi accessible que le type de classe lui-même.
Interfaces	Les interfaces de base explicites d'un type d'interface doivent être au moins aussi accessibles que le type d'interface lui-même.
Délégués	Le type de retour et les types de paramètres d'un type délégué doivent être au moins aussi accessibles que le type délégué lui-même.
Constantes	Le type d'une constante doit être au moins aussi accessible que la constante elle-même.
Champs	Le type d'un champ doit être au moins aussi accessible que le champ lui-même.
Méthodes	Le type de retour et les types de paramètres d'une méthode doivent être au moins aussi accessibles que la méthode elle-même.
Propriétés	Le type d'une propriété doit être au moins aussi accessible que la propriété elle-même.
Événements	Le type d'un événement doit être au moins aussi accessible que l'événement lui-même.
Indexeurs	Le type et les types de paramètres d'un indexeur doivent être au moins aussi accessibles que l'indexeur lui-même.
Opérateurs	Le type de retour et les types de paramètres d'un opérateur doivent être au moins aussi accessibles que l'opérateur lui-même.
Constructeurs	Les types de paramètres d'un constructeur doivent être au moins aussi accessibles que le constructeur lui-même.

Modification de visibilité

Rappelons les classiques modificateurs de visibilité des **variables** et des **méthodes** dans les langages orientés objets, dont C# dispose :

Les mots clef (modularité public-privé)

par défaut **(aucun mot clef)**	Les variables et les méthodes d'une classe non précédées d'un mot clef sont **private** et ne sont visibles que dans la classe seulement.
public	Les variables et les méthodes d'une classe précédées du mot clef **public** sont visibles par toutes les classes de tous les modules.
private	Les variables et les méthodes d'une classe précédées du mot clef **private** ne sont visibles que dans la classe seulement.
protected	Les variables et les méthodes d'une classe précédées du mot clef **protected** sont visibles par toutes les classes dérivées de cette classe.
internal	Les variables et les méthodes d'une classe précédées du mot clef **internal** sont visibles par toutes les classes

	inclues dans le même assembly.

Les attributs d'accessibilité **public, private, protected** sont identiques à ceux de Delphi et Java, pour les classes nous donnons ci-dessous des informations sur leur utilisation.

L'attribut **internal** joue à peu près le rôle (au niveau de l'assembly) des classes Java déclarées sans mot clef dans le même package, ou des classes Delphi déclarées dans la même unit (classes amies). Toutefois pour des raisons de sécurité C# ne possède pas la notion de classe amie.

1. Les classes : des nouveaux types

Rappelons un point fondamental déjà indiqué : tout programme C# contient une ou plusieurs classes précédées ou non d'une déclaration d'utilisation d'autres classes contenues dans des bibliothèques (clause **using**) ou dans un package complet composé de nombreuses classes.

La notion de module en C# est représentée par l'espace de noms (clause **namespace**) semblable au package Java, en C# vous pouvez omettre de spécifier un namespace, par défaut les classes déclarées le sont automatiquement dans un espace 'sans nom' (généralement qualifié de global) et tout identificateur de classe déclaré dans cet espace global sans nom est disponible pour être utilisé dans un espace de noms nommé.

Contrairement à Java, en C# les classes non qualifiées par un modificateur de visibilité (déclarées sans rien devant) sont **internal**.

Delphi	Java	C#

Unit Biblio**;**
interface
 	
// les déclarations des classes
 	
implementation
 	
// les implémentations des classes
 	
end.
	package Biblio;
	
	// les déclarations et implémentation des classes
	
	si pas de nom de package alors automatiquement dans :
	package java.lang;
		namespace Biblio
		{
		
		// les déclarations et implémentation des classes
		
		}
		si pas de nom d'espace de noms alors automatiquement dans l'espace global.

1.1 Déclaration d'une classe

En C#, nous n'avons pas comme en Delphi, une partie déclaration de la classe et une partie implémentation séparées l'une de l'autre. La classe avec ses attributs et ses méthodes sont déclarés et implémentés à un seul endroit comme en Java.

Delphi	Java	C#
interface **uses** biblio; **type** Exemple = **class** x : real; y : integer; **function** F1(a,b:integer): real; **procedure** P2; **end**; **implementation** **function** F1(a,b:integer): real; **begin** **.....code de F1** **end**; **procedure** P2; **begin** **.....code de P2** **end**; **end**.	**import** biblio; **class** Exemple **{** **float** x; **int** y; **float** F1(**int** a, **int** b) **{** **.....code de F1** **}** void P2( ) **{** **.....code de P2** **}** **}**	**using** biblio; **namespace** Machin **{** **class** Exemple **{** **float** x; **int** y; **float** F1(**int** a, **int** b) **{** **.....code de F1** **}** void P2( ) **{** **.....code de P2** **}** **}** **}**

1.2 Une classe est un type en C#

Comme en Delphi et Java, une classe C# peut être considérée comme un nouveau type dans le programme et donc des variables d'objets peuvent être déclarées selon ce nouveau "type".

Une déclaration de programme comprenant 3 classes :

Delphi	Java	C#
interface **type** Un = **class** ... **end**; Deux = **class** ... **end**; Appli3Classes = **class** x : Un; y : Deux; **public** **procedure** main; **end**; **implementation** **procedure** Appli3Classes.main; **var** x : Un; y : Deux; **begin** ... **end**; **end.**	**class** Appli3Classes **{** Un x; Deux y; **public static void** main(String [ ] arg) **{** Un x; Deux y; ... **}** **}** **class** Un **{** ... **}** **class** Deux **{** ... **}**	**class** Appli3Classes **{** Un x; Deux y; **static void** Main(String [ ] arg) **{** Un x; Deux y; ... **}** **}** **class** Un **{** ... **}** **class** Deux **{** ... **}**

1.3 Toutes les classes ont le même ancêtre - héritage

Comme en Delphi et en Java, toutes les classes C# dérivent automatiquement d'une seule et même classe ancêtre : la classe **Object**. En C# le mot-clef pour indiquer la dérivation (héritage) à partir d'une autre classe est le symbole deux points **':'**, lorsqu'il est omis c'est donc que la classe hérite automatiquement de la classe **Object** :

Les deux déclarations de classe ci-dessous sont équivalentes :

Delphi	Java	C#
type Exemple = **class** (TObject) ..... **end**;	**class** Exemple **extends** Object **{** ..... **}**	**class** Exemple **:** Object **{** ..... **}**
type Exemple = **class** ..... **end**;	**class** Exemple **{** ..... **}**	**class** Exemple **{** ..... **}**

L'héritage en C# est tout à fait classiquement de l'**héritage simple** comme en Delphi et en Java. Une classe fille qui dérive d'une seule classe mère, hérite de sa classe mère toutes ses méthodes et tous ses champs. En C# la syntaxe de l'héritage fait intervenir le symbole clef **':'**, comme dans "**class** Exemple **:** Object".

Une déclaration du type :
class ClasseFille **:** ClasseMere {
}
signifie que la classe ClasseFille dispose de tous les attributs et de toutes les méthodes de la classe ClasseMere.

Comparaison héritage :

Delphi	Java	C#

type
ClasseMere = **class**
// champs de ClasseMere
// méthodes de ClasseMere
end;
 	
ClasseFille = **class (** ClasseMere **)**
// hérite des champs de ClasseMere
// hérite des méthodes de ClasseMere
end;
	class ClasseMere
	{
	// champs de ClasseMere
	// méthodes de ClasseMere
	}
	
	class ClasseFille **extends** ClasseMere
	{
	// hérite des champs de ClasseMere
	// hérite des méthodes de ClasseMere
	}
		class ClasseMere
		{
		// champs de ClasseMere
		// méthodes de ClasseMere
		}
		
		class ClasseFille **:** ClasseMere
		{
		// hérite des champs de ClasseMere
		// hérite des méthodes de ClasseMere
		}

Bien entendu une classe fille peut définir de nouveaux champs et de nouvelles méthodes qui lui sont propres.

1.4 Encapsulation des classes

La visibilité et la protection des classes en Delphi est apportée par le module **Unit** où toutes les classes sont visibles dans le module en entier et dès que la unit est utilisée les classes sont visibles partout. Il n'y a pas de possibilité d'imbriquer une classe dans une autre.

En C#, nous avons la possibilité d'***imbriquer*** des classes dans d'autres classes (classes internes), par conséquent la ***visibilité de bloc s'applique aussi aux classes.***

Remarque

La notion de classe interne de C# (qui n'existe pas en Delphi) est sensiblement différente de celle de Java (pas de classe locale et pas de classe anonyme mais des méthodes anonymes), elle correspond à la notion de **classe membre** de Java.

Mots clefs pour la protection des classes et leur visibilité :

- Une classe C# peut se voir attribuer un modificateur de comportement sous la forme d'un mot clef devant la déclaration de classe. Par défaut si aucun mot clef n'est indiqué la classe est visible dans tout le namespace dans lequel elle est définie. Il y a 4 qualificateurs possibles pour modifier le comportement de visibilité d'une classe selon sa position (imbriquée ou non) : **public, private, protected, internal** (dénommés modificateurs d'accès) et **abstract** (qualificateur d'abstraction pouvant être associé à l'un des 3 autres modificateurs d'accès). On rappelle que sans qualificateur **public, private, internal** ou **protected**, une classe C# est automatiquement **internal.**
- Le nom du fichier source dans lequel plusieurs classes C# sont stockées n'a aucun rapport avec le nom d'une des classes déclarées dans le texte source, il est laissé au libre choix du développeur et peut éventuellement être celui d'une classe du namespace etc...

Tableau des possibilités fourni par microsoft :

Membres de	Accessibilité des membres par défaut	Accessibilité déclarée autorisée du membre
enum	public	Aucune
class	private	public protected internal private protected internal
interface	public	Aucune
struct	private	public internal private

Attention

Par défaut dans une classe tous les membres sans qualificateur de visibilité (classes internes

inclues) sont **private**.

C#	Explication
mot clef **abstract** : **abstract class** ApplicationClasse1 { ... }	classe abstraite **non instanciable**. Aucun objet ne peut être créé.
mot clef **public** : **public class** ApplicationClasse2 { ... }	classe visible **par n'importe quel** programme d'un autre namespace
mot clef **protected** : **protected class** ApplicationClasse3 { ... }	classe visible **seulement** par toutes les autres classes héritant de la classe conteneur de cette classe.
mot clef **internal** : **internal class** ApplicationClasse4 { ... }	classe visible **seulement** par toutes les autres classes du même assembly.
mot clef **private** : **private class** ApplicationClasse5 { ... }	classe visible **seulement** par toutes les autres classes du même namespace où elle est définie.
pas de mot clef : **class** ApplicationClasse6 { ... } - sauf si c'est une classe interne	qualifiée **internal** -si c'est une classe interne elle est alors qualifiée **private**.

Nous remarquons donc qu'une classe dès qu'elle est déclarée dans l'espace de noms est toujours visible dans tout l'assembly**,** que le mot clef **internal** soit présent ou non. Les mots clefs **abstract** et **protected** n'ont de l'influence que pour l'héritage.

Remarque

La notion de classe **sealed** en C# correspond strictement à la notion de classe **final** de Java : ce sont des **classes non héritables**.

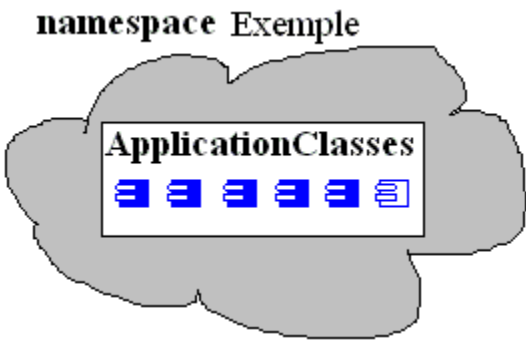
Nous étudions ci-après la visibilité des classes précédentes dans deux contextes différents.

1.5 Exemple de classes imbriquées dans une autre classe

Dans le premier contexte, ces six classes sont utilisées en étant **intégrées** (imbriquées) à une classe publique.

La classe ApplicationClasses ***:***

C#	Explication

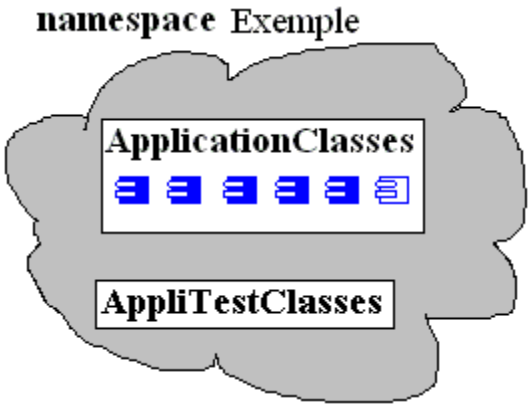



```
namespace Exemple
{
 public class ApplicationClasses {
  abstract class ApplicationClasse1 { ... }
  public class ApplicationClasse2 { ... }
  protected class ApplicationClasse3 { ... }
  internal class ApplicationClasse4 { ... }
  private class ApplicationClasse5 { ... }
  class ApplicationClasse6 { ... }
 }
}
```

Ces 6 "sous-classes" sont visibles ou non, à partir de l'accès à la classe englobante "ApplicationClasses", elles peuvent donc être utilisées dans tout programme qui utilise la classe "ApplicationClasses".

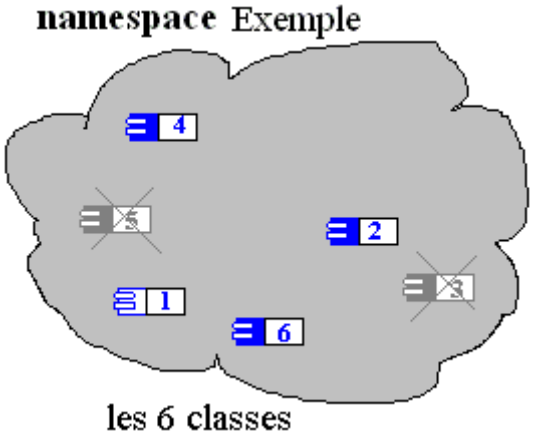
Par défaut la classe ApplicationClasse6 est ici **private**.

Un programme utilisant la classe ApplicationClasses ***:***

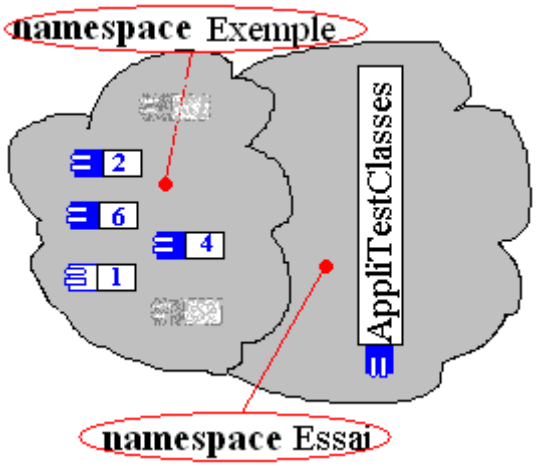
C#	Explication
	Le programme de gauche "**class** AppliTestClasses" *utilise* la classe précédente ApplicationClasses et ses sous-classes. La notation uniforme de chemin de classe est standard.
namespace Exemple **{** **class** AppliTestClasses **{** ApplicationClasses.ApplicationClasse2 a2; **}** **}**	Seule la classe interne ApplicationClasse2 est visible et permet d'instancier un objet.

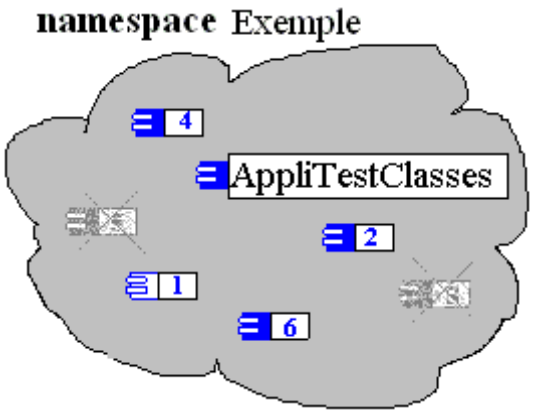
1.6 Même exemple de classes non imbriquées situées dans le même espace de noms

Dans ce second exemple, ces mêmes 6 classes sont utilisées en étant **incluses** dans le même namespace.

C#	Explication
 **namespace** Exemple **{** **abstract class** ApplicationClasse1 { ... } **public class** ApplicationClasse2 { ... } **protected class** ApplicationClasse3 { ... } **internal class** ApplicationClasse4 { ... } **private class** ApplicationClasse5 { ... } **class** ApplicationClasse6 { ... } **}**	Une classe dans un espace de nom ne peut pas être qualifiée **protected** ou **private** (si elle est imbriquée comme ci-haut cela est possible): *Les classes ApplicationClasse3 et ApplicationClasse5 ne peuvent donc pas faire partie du même **namespace** Exemple.* La classe ApplicationClasse1 est ici abstraite et **public**, donc visible. La classe ApplicationClasse2 est **public**, donc visible. La classe ApplicationClasse4 n'est visible que dans le même assembly. Par défaut la classe ApplicationClasse6 est ici **public**, donc visible.

Un programme AppliTestClasses ***utilisant ces 4 classes :***

C# dans deux **namespace** différents	Explication
 **using** Exemple; **namespace** Essai **{** **class** AppliTestClasses**{** ApplicationClasse2 a2; ApplicationClasse6 a6; **}** **}**	Le programme de gauche "**class** AppliTestClasses" utilise les classes qui composent le **namespace** Exemple. Cette classe AppliTestClasses est définie dans un autre **namespace** dénommé Essai qui est supposé ne pas faire partie du même assembly que le **namespace** Exemple. Elle ne voit donc pas la classe ApplicationClasse4 (visible dans le même assembly seulement). Si l'on veut instancier des objets, seules les classes ApplicationClasse2 et ApplicationClasse6 sont de bonnes candidates, car la classe ApplicationClasse1 bien qu'elle soit visible est abstraite.

C# dans le même **namespace**	Explication
 **namespace** Exemple **{** **class** AppliTestClasses**{** ApplicationClasse1 a1; ApplicationClasse2 a2; ApplicationClasse4 a4; ApplicationClasse6 a6; **}** **}**	Le programme de gauche "**class** AppliTestClasses" utilise les classes qui composent le **namespace** Exemple. La classe AppliTestClasses est définie dans le même **namespace** Exemple que les 4 autres classes. Toutes les classes du même namespace sont visibles entre elles. Ici toutes les 4 classes sont visibles pour la classe AppliTestClasses.

Remarque pratique :

Selon sa situation imbriquée ou non imbriquée, une classe peut ou ne peut pas être qualifiée par les divers modificateurs de visibilité. En cas de doute le compilateur fournit un diagnostique clair, comme ci-dessous :

[C# Erreur] Class.cs(nn): Les éléments namespace ne peuvent pas être déclarés explicitement comme private, protected ou protected internal.

1.7 Méthodes abstraites

Le mot clef **abstract** est utilisé pour représenter **une classe ou une méthode abstraite**. Quel est l'intérêt de cette notion ? Avoir des modèles génériques permettant de définir ultérieurement des actions spécifiques.

Une méthode déclarée en abstract dans une classe mère :

- N'a pas de corps de méthode.
- N'est pas exécutable.

- Doit obligatoirement être redéfinie dans une classe fille.

Une méthode **abstraite** n'est qu'une **signature** de méthode sans implémentation dans la classe.

Exemple de méthode abstraite :

```
class Etre_Vivant {  }
```

La classe Etre_Vivant est une classe mère générale pour les êtres vivants sur la planète, chaque catégorie d'être vivant peut être représentée par une classe dérivée (classe fille de cette classe) :

```
class Serpent : Etre_Vivant {  }

class Oiseau : Etre_Vivant {   }

class Homme : Etre_Vivant {   }
```

Tous ces êtres se déplacent d'une manière générale, donc une méthode SeDeplacer est commune à toutes les classes dérivées, toutefois chaque espèce exécute cette action d'une manière différente et donc on ne peut pas dire que se déplacer est une notion concrète mais une notion abstraite que chaque sous-classe précisera concrètement.

En C#, les méthodes abstraites sont automatiquement virtuelles, elles ne peuvent être déclarées que **public** ou **protected**, enfin elles doivent être redéfinies avec le qualificateur **override**. Ci-dessous deux déclarations possibles pour le déplacement des êtres vivants :

```
abstract class Etre_Vivant {
  public abstract void SeDeplacer( );
}

class Serpent : Etre_Vivant {
  public override void SeDeplacer( ) {
    //.....en rampant
  }
}

class Oiseau : Etre_Vivant {
  public override void SeDeplacer( ) {
    //.....en volant
  }
}

class Homme : Etre_Vivant {
  public override void SeDeplacer( ) {
    //.....en marchant
  }
}
```

```
abstract class Etre_Vivant {
  protected abstract void SeDeplacer( );
}

class Serpent : Etre_Vivant {
  protected override void SeDeplacer( ) {
    //.....en rampant
  }
}

class Oiseau : Etre_Vivant {
  protected override void SeDeplacer( ) {
    //.....en volant
  }
}

class Homme : Etre_Vivant {
  protected override void SeDeplacer( ) {
    //.....en marchant
  }
}
```

Comparaison de déclaration d'abstraction de méthode en Delphi et C# :

Delphi	C#
type Etre_Vivant = **class**	**abstract class** Etre_Vivant **{** **public abstract void** SeDeplacer( );

procedure SeDeplacer;**virtual**; **abstract** ; **end**; Serpent = **class** (Etre_Vivant) **procedure** SeDeplacer; **override**; **end**; Oiseau = **class** (Etre_Vivant) **procedure** SeDeplacer; **override**; **end**; Homme = **class** (Etre_Vivant) **procedure** SeDeplacer; **override**; **end**;	**}** **class** Serpent **:** Etre_Vivant **{** **public override void** SeDeplacer( ) **{** //.....en rampant **}** **}** **class** Oiseau **:** Etre_Vivant **{** **public override void** SeDeplacer( ) **{** //.....en volant **}** **}** **class** Homme **:** Etre_Vivant **{** **public override void** SeDeplacer( ) **{** //.....en marchant **}** **}**

En C# une méthode **abstraite** est une méthode **virtuelle** n'ayant pas d'implémentation dans la classe où elle est déclarée. Son implémentation est déléguée à une classe dérivée. Les méthodes abstraites doivent être déclarées en spécifiant la directive **abstract** .

1.8 Classe abstraite, Interface

Classe abstraite

Comme nous venons de le voir dans l'exemple précédent, une classe C# peut être précédée du mot clef **abstract**, ce qui signifie alors que cette classe est abstraite, nous avons les contraintes de définition suivantes pour une classe abstraite en C# :

Si une classe contient au moins une méthode **abstract**, elle doit impérativement être déclarée en classe **abstract** elle-même. C'est ce que nous avons écrit au paragraphe précédent pour la classe Etre_Vivant que nous avons déclarée **abstract** parce qu'elle contenait la méthode abstraite SeDeplacer.

Une classe **abstract** ne peut pas être instanciée directement, seule une classe dérivée (sous-classe) qui redéfinit obligatoirement toutes les méthodes **abstract** de la classe mère peut être instanciée.

Conséquence du paragraphe précédent, une classe dérivée qui redéfinit toutes les méthodes **abstract** de la classe mère sauf une (ou plus d'une) ne peut pas être instanciée et subit la même règle que la classe mère : elle contient au moins une méthode abstraite donc elle est aussi une classe abstraite et doit donc être déclarée en **abstract**.

Une classe **abstract** peut contenir des méthodes non abstraites et donc implantées dans la classe. Une classe **abstract** peut même ne pas contenir du tout de méthodes abstraites, dans ce cas une classe fille n'a pas la nécessité de redéfinir les méthodes de la classe mère pour être instanciée.

Delphi contrairement à C# et Java, ne possède pas à ce jour le modèle de la classe abstraite, seule la version Delphi2005 pour le Net Framework possède les mêmes caractéristiques que C#.

Interface

Lorsqu'une classe est déclarée en **abstract** et que toutes ses méthodes sont déclarées en **abstract**, on appelle en C# une telle classe une **Interface**.

Rappel classes abstraites-interfaces

- Les interfaces ressemblent aux classes abstraites sur un seul point : elles contiennent des membres **expliquant certains comportements sans les implémenter**.
- Les classes abstraites et les interfaces se différencient principalement par le fait qu'**une classe peut implémenter un nombre quelconque d'interfaces**, alors qu'une classe abstraite ne peut hériter que d'**une seule classe** abstraite ou non.

Vocabulaire et concepts :

- Une interface est un contrat, elle peut contenir des **propriétés**, des **méthodes** et des **événements** mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une interface **ne** peut **pas** contenir des méthodes déjà implémentées.
- Une interface doit contenir des méthodes **non** implémentées.
- Une interface est héritable.
- On peut contsruire une hiérarchie d'interfaces.
- Pour pouvoir construire un objet à partir d'une interface, il faut définir une classe non abstraite implémentant **toutes** les méthodes de l'interface.

Une classe peut implémenter plusieurs interfaces. Dans ce cas nous avons une excellente alternative à l'héritage multiple.

Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui ne devraient plus être modifiés. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

Les variables de types interface respectent les mêmes règles de **transtypage** que les variables de types classe.

Les **objets** de type classe **clA** peuvent être transtypés et **referencés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

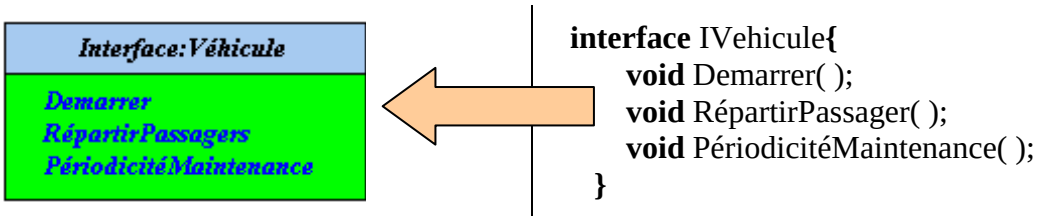
Si vous voulez utiliser la notion d'interface pour fournir un polymorphisme à une famille de classes, elles doivent toutes implémenter cette interface, comme dans l'exemple ci-dessous.

Exemple :

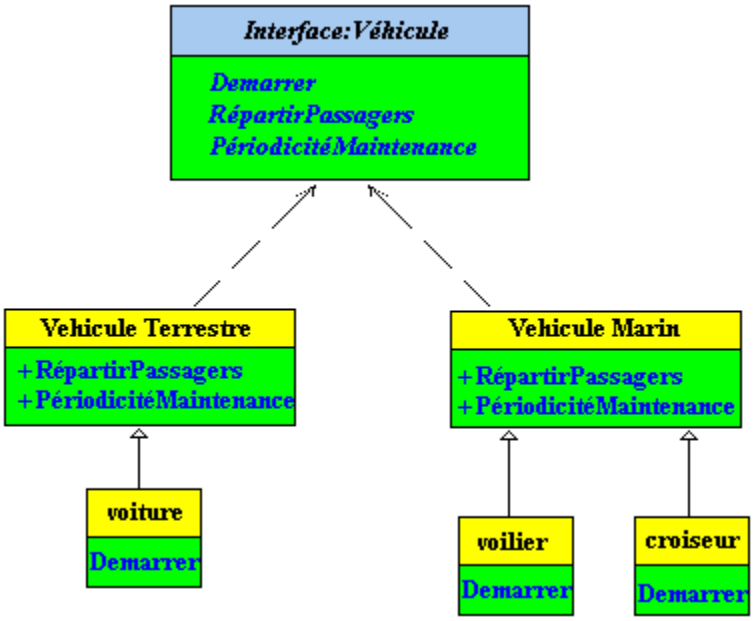
l'interface ***Véhicule*** définissant 3 méthodes (abstraites) **Demarrer**, **RépartirPassagers** de

répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), et **PériodicitéMaintenance** renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de km ou miles parcourus, du nombre d'heures d'activités,...)

Soit l'interface ***Véhicule*** définissant ces 3 méthodes :



Soient les deux classes **Véhicule terrestre** et **Véhicule marin**, qui implémentent partiellement chacune l'interface ***Véhicule*** , ainsi que trois classes **voiture**, **voilier** et **croiseur** héritant de ces deux classes :



- Les trois méthodes de l'interface ***Véhicule*** sont abstraites et publiques par définition.

- Les classes **Véhicule terrestre** et **Véhicule marin** sont abstraites, car la méthode abstraite **Demarrer** de l'interface ***Véhicule*** n'est pas implémentée elle reste comme "modèle" aux futures classes. C'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

> Dans cette vision de la hiérarchie on a supposé que les classes abstraites **Véhicule terrestre** et **Véhicule marin** savent comment répartir leur éventuels passagers et quand effectuer une maintenance du véhicule.
>
> Les classes **voiture**, **voilier** et **croiseur** , n'ont plus qu'à implémenter chacune son propre comportement de démarrage.

Une interface C# peut être qualifiée par un des 4 modificateur **public, protected, internal, private**.

Contrairement à Java, une classe abstraite C# qui implémente une interface doit **obligatoirement** déclarer **toutes** les méthodes de l'interface, celles qui ne sont pas implémentées dans la classe abstraite doivent être déclarées abstract. C'est le cas dans l'exemple ci-dessous pour la méthode abstraite **Demarrer** , nous proposons deux écritures possibles pour cette hiérarchie de classe :

C# méthode **abstraite** sans corps	**C#** méthode **virtuelle** à corps vide

interface IVehicule{
void Demarrer( );
void RépartirPassager( );
void PériodicitéMaintenance( );
}
 	
abstract class Terrestre : IVehicule {
public abstract void Demarrer( );
public virtual void RépartirPassager( ){...}
public virtual void PériodicitéMaintenance( ){...}
}
 	
class Voiture : Terrestre {
override public void Demarrer( ){...}
}
 	
abstract class Marin : IVehicule {
public abstract void Demarrer( );
public virtual void RépartirPassager( ){...}
public virtual void PériodicitéMaintenance( ){...}
}
 	
class Voilier : Marin {
override public void Demarrer( ){...}
}
class Croiseur : Marin {
override public void Demarrer( ){...}
}
	interface IVehicule{
	void Demarrer( );
	void RépartirPassager( );
	void PériodicitéMaintenance( );
	}
	
	abstract class Terrestre : IVehicule {
	public virtual void Demarrer( ){}
	public virtual void RépartirPassager( ){...}
	public virtual void PériodicitéMaintenance( ){...}
	}
	
	class Voiture : Terrestre {
	override public void Demarrer( ){...}
	}
	
	abstract class Marin : IVehicule {
	public virtual void Demarrer( ) {}
	public virtual void RépartirPassager( ){...}
	public virtual void PériodicitéMaintenance( ){...}
	}
	
	class Voilier : Marin {
	override public void Demarrer( ){...}
	}
	class Croiseur : Marin {
	override public void Demarrer( ){...}
	}

Les méthodes RépartirPassagers, PériodicitéMaintenance et Demarrer sont implantées en **virtual** , soit comme des méthodes à liaison dynamique, afin de laisser la possibilité pour des classes enfants de redéfinir ces méthodes.

Remarque :

Attention le qualificateur **new** ne peut masquer que des membres non abstract. Un membre abstract doit impérativement être redéfini (implémenté) par le qualificateur **override**, car ce genre de membre est implicitement **virtual** en C#.

Soit à titre de comparaison, les deux mêmes écritures en Java de ces classes :

Java , méthode abstraite sans corps

```
interface IVehicule{
    void Demarrer( );
    void RépartirPassager( );
    void PériodicitéMaintenance( );
}

abstract class Terrestre  implements IVehicule {
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voiture extends Terrestre {
    public void Demarrer( ){};
}

abstract class Marin implements IVehicule {
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voilier extends Marin {
    public void Demarrer( ){};
}

class Croiseur extends Marin {
    public void Demarrer( ){};
}
```

Java méthode virtuelle à corps vide

```
interface IVehicule{
    void Demarrer( );
    void RépartirPassager( );
    void PériodicitéMaintenance( );
}

abstract class Terrestre  implements IVehicule {
    public void Demarrer( ){};
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voiture extends Terrestre {
    public void Demarrer( ){};
}

abstract class Marin implements IVehicule {
    public void Demarrer( ){};
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voilier extends Marin {
    public void Demarrer( ){};
}

class Croiseur extends Marin {
    public void Demarrer( ){};
}
```

2. Les objets : des références ou des valeurs

Les classes sont des descripteurs d'objets, les objets sont les agents effectifs et "vivants" implantant les actions d'un programme. Les objets dans un programme ont une vie propre :

- Ils naissent (ils sont créés ou alloués).
- Ils agissent (ils s'envoient des messages grâce à leurs méthodes).
- Ils meurent (ils sont désalloués, automatiquement en C#).

C'est dans le segment de mémoire du CLR de .NetFramework que s'effectuent l'allocation et la désallocation d'objets.

Objets type valeur

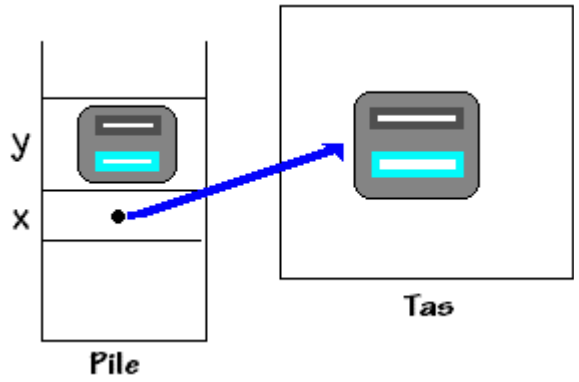
Les classes encapsulant les types élémentaires dans **.NET Framework** sont des classes de type valeur du genre **structures**. Dans le CLS une classe de type valeur est telle que les allocations d'objets de cette classe se font directement dans la pile et non dans le tas, il n'y a donc pas de référence pour un objet de type valeur et lorsqu'un objet de type valeur est passé comme paramètre il est **passé par valeur**.

Dans **.NET Framework** les classes-structures de type valeur sont déclarées comme structures et ne sont pas dérivables

Objets type référence

Le principe d'allocation et de représentation des objets type référence en C# est identique à celui de Delphi il s'agit de la référence, qui est une encapsulation de la notion de pointeur. Dans **.NET Framework** les classes de type référence sont déclarées comme des classes classiques et sont dérivables.

Afin d'éclairer le lecteur prenons par exemple un objet **x instancié à partir d'une classe de type référence** et un objet **y instancié à partir d'un classe de type valeur** contenant les mêmes membres que la classe par référence. Ci-dessous le schéma d'allocation de chacun des deux catégories d'objets :



Pour les types valeurs, la gestion mémoire des objets est **classiquement celle de la pile dynamique**, un tel objet se comporte comme une variable locale de la méthode dans laquelle il est instancié et ne nécessite pas de gestion supplémentaire. Seuls les objets type référence instanciés sur le tas, nécessitent une gestion mémoire spéciale que nous détaillons ci-après (dans un

programme C# les types références du développeur représentent près de 99% des objets du programme).

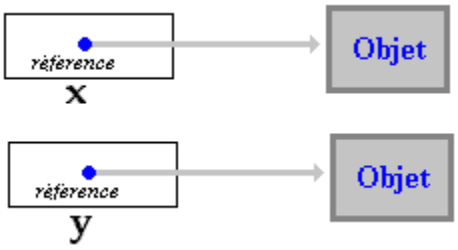
2.1 Modèle de la référence en C#

Rappelons que dans le modèle de la référence chaque objet (représenté par un identificateur de variable) est caractérisé par un couple (référence, bloc de données). Comme en Delphi, C# décompose l'**instanciation** (allocation) d'un objet en deux étapes :

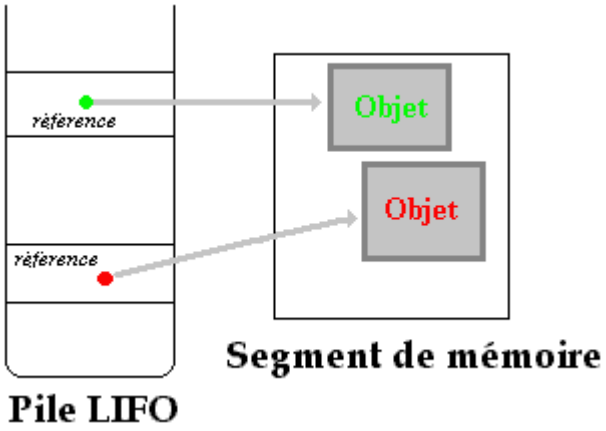
- La déclaration d'identificateur de variable typée qui contiendra la référence,
- la création de la structure de données elle-même (bloc objet de données) avec **new**.

Delphi	C#
type **Un** = **class** **.....** **end**; *// la déclaration :* **var** **x , y** : **Un**; .... *// la création :* **x** := Un.create ; **y** := Un.create ;	**class Un** **{** ... **}** *// la déclaration :* **Un x , y ;** .... *// la création :* **x** = **new Un**( ); **y** = **new Un**( );

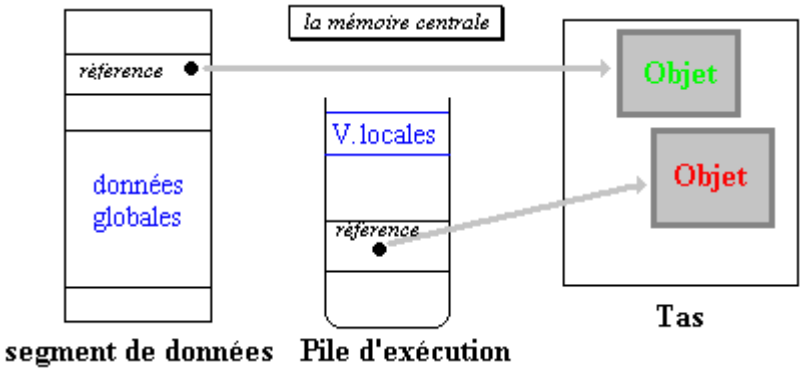
Après exécution du pseudo-programme précédent, les variables x et y contiennent chacune une référence (adresse mémoire) vers un bloc objet différent:



Un programme C# est fait pour être exécuté par l'environnement CLR de .NetFramework. Deux objets C# seront instanciés dans le CLR de la manière suivante :

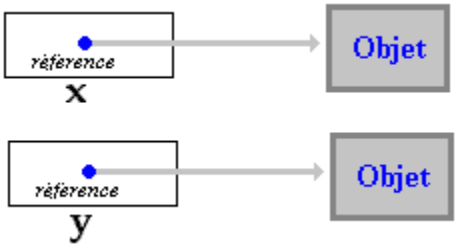


Attitude à rapprocher pour comparaison, à celle dont **Delphi** gère les objets dans une pile d'exécution de type LIFO et un tas :

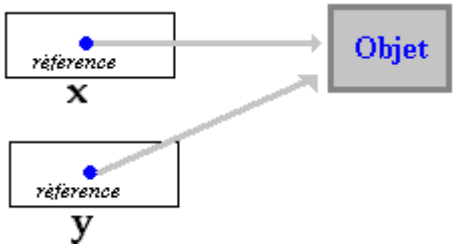


Attention à l'utilisation de l'affectation entre variables d'objets dans le modèle de représentation par référence. L'affectation x = y ne recopie pas le bloc objet de données de y dans celui de x, mais seulement la référence (l'adresse) de y dans la référence de x. Visualisons cette remarque importante :

Situation au départ, avant affectation



Situation après l'affectation " x = y "



En C#, la désallocation étant automatique, le bloc de données objet qui était référencé par **y** avant

l'affectation, n'est pas perdu, car le garbage collector se charge de restituer la mémoire libérée au **segment de mémoire** du CLR

2.2 Les constructeurs d'objets références ou valeurs

Un constructeur est une **méthode spéciale** d'une classe dont la seule fonction est d'**instancier** un objet (créer le bloc de données). Comme en Delphi une **classe C# peut posséder plusieurs constructeur**s, il est possible de pratiquer des initialisations d'attributs dans un constructeur. Comme toutes les méthodes, un constructeur peut avoir ou ne pas avoir de paramètres formels.

- Si vous ne déclarez pas de constructeur spécifique pour une classe, **par défaut** C# attribue automatiquement un constructeur sans paramètres formels, portant le même nom que la classe. A la différence de Delphi où le nom du constructeur est quelconque, en C# le(ou les) **constructeur doit obligatoirement porter le même nom que la classe** (majuscules et minuscules comprises).
- Un constructeur d'objet d'une classe n'a d'intérêt que s'il est visible par tous les programmes qui veulent instancier des objets de cette classe, c'est pourquoi l'on mettra toujours le mot clef **public** devant la déclaration du constructeur.
- Un constructeur est une méthode spéciale dont la fonction est de créer des objets, dans son en-tête il n'a pas de type de retour et le mot clef **void** n'est pas non plus utilisé !

Soit une classe dénommée Un dans laquelle, comme nous l'avons fait jusqu'à présent nous n'indiquons aucun constructeur spécifique :

```
class Un  {
   int a;
 }
```

Automatiquement C# attribue un constructeur public à cette classe **public** Un ( ). C'est comme si C# avait introduit dans votre classe à votre insu , une nouvelle méthode dénommée Un. Cette méthode "cachée" n'a aucun paramètre et aucune instruction dans son corps. Ci-dessous un exemple de programme C# correct illustrant ce qui se passe :

```
class Un  {
  public Un ( ) { }
  int a;
 }
```

- **Vous pouvez programmer et personnaliser vos propres constructeurs.**
- **Une classe C# peut contenir plusieurs constructeurs dont les en-têtes diffèrent uniquement par la liste des paramètres formels.**

Exemple de constructeur avec instructions :

C#	Explication
class Un **{ public** Un ( ) **{** a = 100 **}** **int** a; **}**	Le constructeur **public** Un sert ici à initialiser à 100 la valeur de l'attribut "**int** a" de chaque objet qui sera instancié.

Exemple de constructeur avec paramètre :

C#	Explication
class Un **{ public** Un (**int** b) **{** a = b; **}** **int** a; **}**	Le constructeur **public** Un sert ici à initialiser la valeur de l'attribut "**int** a" de chaque objet qui sera instancié. Le paramètre **int** b contient cette valeur.

Exemple avec plusieurs constructeurs :

C#	Explication
class Un **{ public** Un (**int** b) **{** a = b; **}** **public** Un ( ) **{** a = 100; **}** **public** Un (**float** b) **{** a = (**int**)b; **}** **int** a; **}**	La classe Un possède 3 constructeurs servant à initialiser chacun d'une manière différente le seul attribut **int** a.

Il est possible de rappeller un constructeur de la classe dans un autre constructeur, pour cela C# utilise comme Java le mot clef **this**, avec une syntaxe différente :

Exemple avec un appel à un constructeur de la même classe:

C#	Explication
class Un **{** **int** a; **public** Un (**int** b) **{** a = b; **}** **public** Un ( )	La classe Un possède 3 constructeurs servant à initialiser chacun d'une manière différente le seul attribut **int** a. **Soit le dernier constructeur :**

{ a = 100; } public Un (float b) { a = (int)b; } public Un (int x , float y) : this(y) { a += 100; } }	public Un (int x , float y) : this(y) { a += 100; } Ce constructeur appelle tout d'abord le constructeur Un (float y) par l'intermédiaire de this(y), puis il exécute le corps de méthode soit : a += 100; Ce qui revient à calculer : a = (int)y + 100 ;

Comparaison Delphi - C# pour la déclaration de constructeurs

Delphi	C#

Un = class
a : integer;
public
constructor creer; overload;
constructor creer (b:integer); overload;
constructor creer (b:real); overload;
end;
 	
implementation
 	
constructor Un.creer; begin
a := 100
end;
constructor Un.creer(b:integer); begin
a := b
end;
constructor Un.creer(b:real); begin
a := trunc(b)
end;
 	
constructor Un.creer(x:integer; y:real);
begin
self.creer(y);
a := a+100;
end;
	class Un
	{
	int a;
	
	public Un ( )
	{ a = 100;
	}
	
	public Un (int b)
	{ a = b;
	}
	
	public Un (float b)
	{ a = (int)b;
	}
	
	public Un (int x , float y) :
	this(y)
	{ a += 100;
	}
	
	}

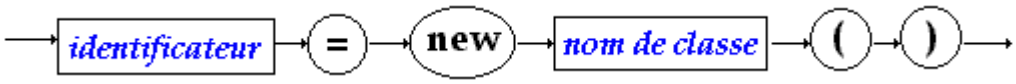
En Delphi un constructeur a un nom quelconque, tous les constructeurs peuvent avoir des noms différents ou le même nom comme en C#.

2.3 Utilisation du constructeur d'objet automatique (par défaut)

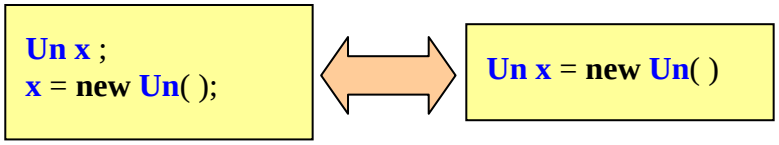
Le constructeur d'objet par défaut de toute classe C# qu'elle soit de type valeur ou de type référence, comme nous l'avons signalé plus haut est une méthode spéciale sans paramètre, l'appel à

cette méthode spéciale afin de construire un nouvel objet répond à une syntaxe spécifique par utilisation du mot clef **new**.

Syntaxe
Pour un constructeur sans paramètres formels, l'instruction d'**instanciation d'un nouvel objet** à partir d'un identificateur de variable déclarée selon un type de classe, s'écrit syntaxiquement ainsi :



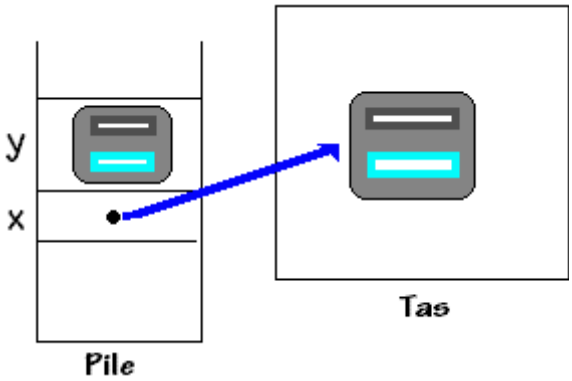
Exemple : (deux façons équivalentes de créer un objet **x** de classe **Un**)



Cette instruction crée dans le segment de mémoire, un nouvel objet de classe **Un** dont la référence (l'adresse) est mise dans la variable **x**, si x est de type référence, ou bien l'objet est directement créé dans la pile et mis dans la variable **x**, si x est de type valeur.

Soit **Un** une classe de type référence et **Deux** une autre classe de type valeur, ci-dessous une image des résulats de l'instanciation d'un objet de chacune de ces deux classes :

Un x = **new** **Un**( ) ;
Deux y = **new** **Deux** ( ) ;



Dans l'exemple ci-dessous, nous utilisons le constructeur par défaut de la classe Un , pour créer deux objets dans une autre classe :

```
class Un
 { ...
 }
```

```
class UnAutre
 {
    // la déclaration :
    Un x , y ;
     ....
   // la création :
    x = new Un( );
    y = new Un( );
 }
```

Un programme de 2 classes, illustrant l'affectation de références :

C#	Explication

class AppliClassesReferences
{
public static void Main(String [ ] arg) **{**
Un x,y ;
x = **new Un**( );
y = **new Un**( );
System.Console.WriteLine("x.a="+x.a);
System.Console.WriteLine"y.a="+y.a);
y = x;
x.a =12;
System.Console.WriteLine("x.a="+x.a);
System.Console.WriteLine("y.a="+y.a);
}
}
class Un
{ int a=10;
}
	Ce programme C# contient deux classes :
	
	class AppliClassesReferences
	et
	class Un
	
	La classe AppliClassesReferences est une classe exécutable car elle contient la méthode main. C'est donc cette méthode qui agira dès l'exécution du programme.

Détaillons les instructions	**Que se passe-t-il à l'exécution ?**

Un x,y ;
x = **new Un**( );
y = **new Un**( );
	Instanciation de 2 objets différents x et y de type **Un**.

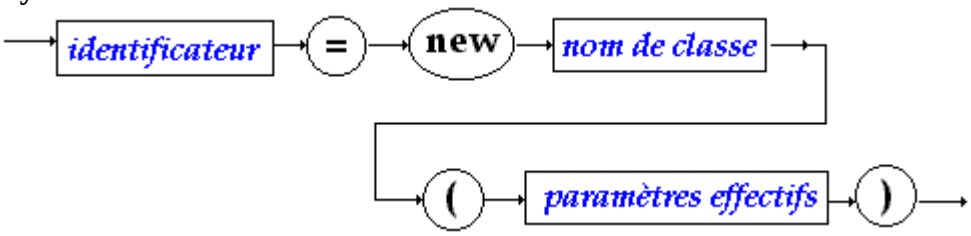
System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a);	*Affichage de :* x.a = 10 y.a = 10
y = x;	La référence de y est remplacée par celle de x dans la variable y (y pointe donc vers le même bloc que x).
x.a =12; System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a);	*On change la valeur de l'attribut a de x, et l'on demande d'afficher les attributs de x et de y :* x.a = 12 y.a = 12 **Comme y pointe vers x, y et x sont maintenant le même objet sous deux noms différents !**

2.4 Utilisation d'un constructeur d'objet personnalisé

L'utilisation d'un constructeur personnalisé d'une classe est semblable à celle du constructeur par

défaut de la classe. La seule différence se trouve lors de l'instanciation : il faut fournir des paramètres effectifs lors de l'appel au constructeur.

Syntaxe



Exemple avec plusieurs constructeurs :

une classe C#	Des objets créés
class Un **{** **int** a ; **public** Un (**int** b) **{** a = b ; **}** **public** Un ( ) **{** a = 100 ; **}** **public** Un (**float** b) **{** a = (**int**)b ; **}** **public** Un (**int** x , **float** y) : **this**(y) **{** a += 100; **}** **}**	**Un** obj1 = **newUn**( ); **Un** obj2 = **new Un**(15); **int** k = 14; **Un** obj3 = **new Un**(k); **Un** obj4 = **new Un**(3.25f); **float** r = -5.6; **Un** obj5 = **new Un**(r); **int** x = 20; **float** y = -0.02; **Un** obj6 = **new Un**(x , y);

2.5 Le mot clef this - cas de la référence seulement

Il est possible de dénommer dans les instructions d'une méthode de classe, un futur objet qui sera instancié plus tard. Le paramètre ou (mot clef) **this** est implicitement présent dans chaque objet instancié et il contient la référence à l'objet actuel. Il joue exactement le même rôle que le mot clef **self** en Delphi. Nous avons déjà vu une de ses utilisations dans le cas des constructeurs.

C#	C# équivalent
class Un **{ public** Un ( ) **{** a = 100; **}** **int** a; **}**	**class Un** **{ public** Un ( ) **{ this**.a = 100; **}** **int** a; **}**

Dans le programme de droite le mot clef **this** fait référence à l'objet lui-même, ce qui dans ce cas est superflu puisque la variable **int** a est un champ de l'objet.

Montrons deux exemples d'utilisation pratique de **this**.

Cas où l'objet est passé comme un paramètre dans une de ses méthodes :

C#	Explications
class Un **{ public Un ( )** **{** a = 100; **}** **public void** methode1(Un x) **{** System.Console.WriteLine("champ a = " + x.a); **}** **public void** methode2(**int** b) **{** a += b; methode1(**this**); **}** **int** a; **}**	La methode1(Un x) reçoit un objet de type Exemple en paramètre et imprime son champ **int** a. La methode2(**int** b) reçoit un entier **int** b qu'elle additionne au champ **int** a de l'objet, puis elle appelle la méthode1 avec comme paramètre l'objet lui-même.

Comparaison Delphi - C# sur cet exemple (similitude complète)

Delphi	C#

Un = **class**
a : integer;
public
constructor creer;
procedure methode1(x:Un);
procedure methode2 (b:integer);
end;
 	
implementation
 	
constructor Un.creer;
begin
a := 100
end;
procedure Un.methode1(x:Un);
begin
showmessage('champ a ='+inttostr(x.a))
end;
procedure Un.methode2 (b:integer);
begin
a := a+b;
methode1(self)
end;
	class Un
	{ public Un ( )
	{ a = 100;
	}
	public void methode1(Un x)
	{
	System.Console.WriteLine("champ a ="+x.a);
	}
	public void methode2(**int** b)
	{ a += b;
	methode1(**this**);
	}
	int a;
	}

Cas où le this sert à outrepasser le masquage de visibilité :

C#	Explications
class Un { **int** a; **public void** methode1(**float** a) **{** a = **this**.a + 7 ; **}** **}**	La methode1(**float** a) possède un paramètre **float** a dont le nom masque le nom du champ **int** a. Si nous voulons malgré tout accéder au champ de l'objet, l'objet étant référencé par **this**, "**this**.a" est donc le champ **int** a de l'objet lui-même.

Comparaison Delphi - C# sur ce second exemple (similitude complète aussi)

Delphi	C#
Un = **class** a : integer; **public** **procedure** methode(a:real); **end**; **implementation** **procedure Un**.methode(a:real);**begin** a = **self**.a + 7 ; **end**;	**class Un** **{** **int** a; **public void** methode(**float** a) **{** a = **this**.a + 7 ; **}** **}**

3. Variables et méthodes

Nous examinons dans ce paragraphe comment C# utilise les variables et les méthodes à l'intérieur d'une classe. Il est possible de modifier des variables et des méthodes d'une classe ceci sera examinée plus loin.

En C#, les champs et les méthodes sont classés en deux catégories :

- **Variables et méthodes de classe**
- **Variables et méthodes d'instance**

3.1 Variables dans une classe en général

Rappelons qu'en C#, nous pouvons déclarer dans un bloc (for, try,...) de nouvelles variables à la condition qu'elles n'existent pas déjà dans le corps de la méthode où elles sont déclarées. Nous les dénommerons : **variables locales de méthode**.

Exemple de variables locales de méthode :

class Exemple **{** **void** calcul (**int** x, **int** y) **{int** a = 100; **for** (**int** i = 1; i<10; i++) **{char** carlu; System.Console.Write("Entrez un caractère : "); carlu = (**char**)System.Console.Read( ); **int** b =15; a =....	La définition **int** a = 100; est locale à la méthode en général La définition **int** i = 1; est locale à la boucle **for**. Les définitions **char** carlu et **int** b sont locales au corps de la boucle **for**.

`  }`
`  }`
`}`	

C# ne connaît pas la notion de variable globale au sens habituel donné à cette dénomination, dans la mesure où toute variable ne peut être définie qu'à l'intérieur d'une classe, ou d'une méthode inclue dans une classe. Donc à part les **variables locales de méthode** définies dans une méthode, C# reconnaît une autre catégorie de variables, ***les variables définies dans une classe mais pas à l'intérieur d'une méthode spécifique***. Nous les dénommerons : **attributs de classes** parce que ces variables peuvent être de deux catégories.

Exemple de attributs de classe :

`class AppliVariableClasse`
`{`
`  float r ;`
 	
`  void calcul ( int x, int y )`
`  { ....`
`  }`
`  int x =100;`
`  int valeur ( char x )`
`  { .....`
`  }`
`  long y;`
`}`
	Les variables **float** r , **long** y et **int** x sont des attributs de classe (ici en fait plus précisément, des variables d'instance).
	
	La position de la déclaration de ces variables n'a aucune importance. Elles sont **visibles dans tout le bloc classe** (c'est à dire visibles par toutes les méthodes de la classe).
	
	Conseil : regroupez les variables de classe au début de la classe afin de mieux les gérer.

Les attributs de classe peuvent être soit de la catégorie des **variables de classe**, soit de la catégorie des **variables d'instance**.

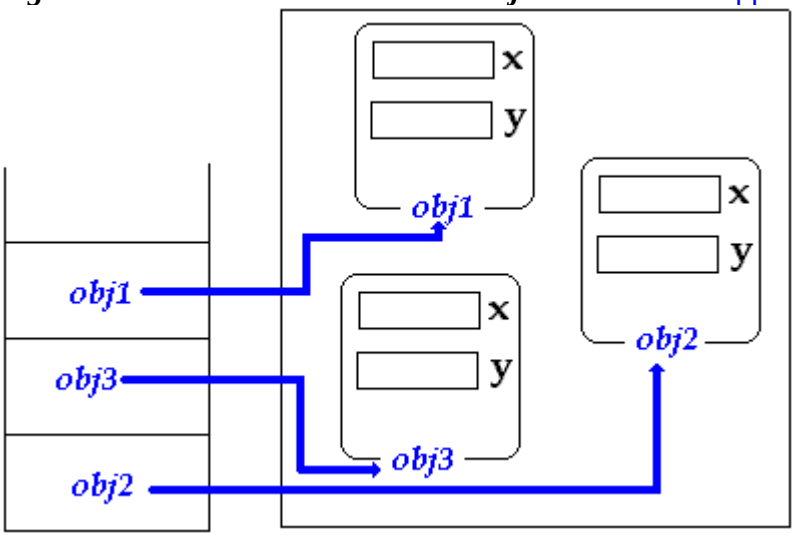
3.2 Variables et méthodes d'instance

C# se comporte comme un langage orienté objet classique vis à vis de ses variables et de ses méthodes. A chaque instanciation d'un nouvel objet d'une classe donnée, la machine CLR enregistre le p-code des méthodes de la classe dans la **zone de stockage** des méthodes, elle alloue dans le **segment de mémoire** autant d'emplacements mémoire pour les variables que d'objet créés. C# dénomme cette catégorie **les variables et les méthodes d'instance**.

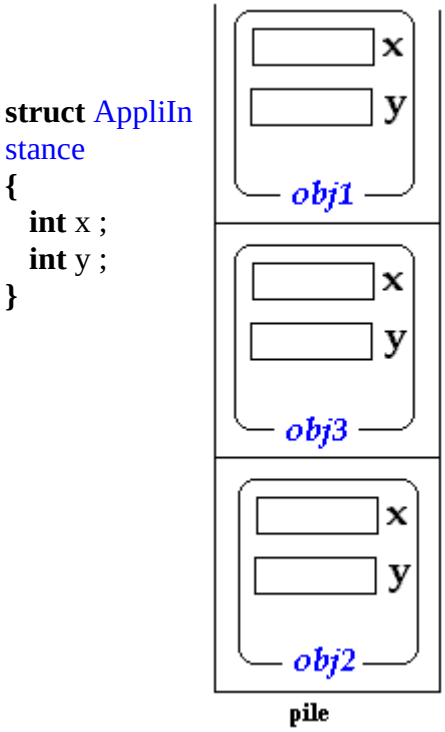
une classe C#	Instanciation de 3 objets

`class AppliInstance`
`{ int x ;`
`  int y ;`
`}`
	`AppliInstance  obj1 = new AppliInstance( );`
	`AppliInstance  obj2 = new AppliInstance( );`
	`AppliInstance  obj3 = new AppliInstance( );`

Segment de mémoire associé à ces 3 objets si la classe AppliInstance ***est de type référence :***



Segment de mémoire associé à ces 3 objets si la classe AppliInstance ***était de type valeur (pour mémoire):***



Un programme C# à 2 classes illustrant l'exemple précédent (classe référence):

Programme C# exécutable
class AppliInstance **{ public int** x = -58 ; **public int** y = 20 ; **}** **class** Utilise

```
{ public static void Main( ) {
    AppliInstance  obj1 = new AppliInstance( );
    AppliInstance  obj2 = new AppliInstance( );
    AppliInstance  obj3 = new AppliInstance( );
    System.Console.WriteLine( "obj1.x = " + obj1.x );
  }
}
```

3.3 Variables et méthodes de classe - static

Variable de classe

On identifie une variable ou une méthode de classe en précédant sa déclaration du mot clef **static**. Nous avons déjà pris la majorité de nos exemples simples avec de tels composants.

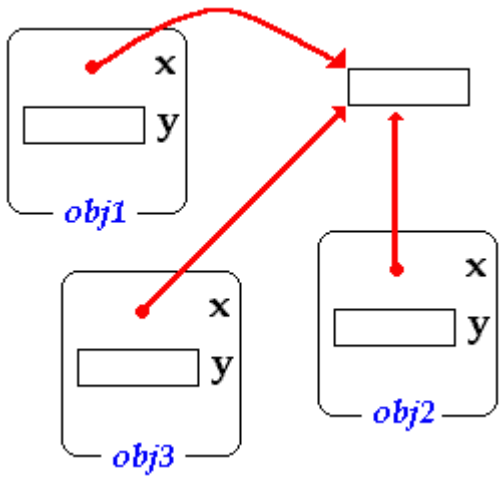
Voici deux déclarations de variables de classe :

```
static int x ;
static int a = 5;
```

Une variable de classe est accessible comme une variable d'instance(selon sa visibilité), mais aussi **sans avoir à instancier un objet de la classe**, uniquement en référençant la variable par le nom de la classe dans la notation de chemin uniforme d'objet.

une classe C#	Instanciation de 3 objets
class AppliInstance { **static int** x ; **int** y ; }	AppliInstance obj1 = **new** AppliInstance( ); AppliInstance obj2 = **new**AppliInstance( ); AppliInstance obj3 = **new**AppliInstance( );

Voici une image du segment de mémoire associé à ces 3 objets :



Exemple de variables de classe :

class ApplistaticVar **{ public static int** x =15 ; **}** **class** UtiliseApplistaticVar **{ int** a ; **void** f( ) **{** a = ApplistaticVar.x ; ..... **}** **}**	La définition "**static int** x =15 ;" crée une variable de la classe ApplistaticVar, nommée x. L'instruction "a = ApplistaticVar.x ;" utilise la variable x comme variable de classe ApplistaticVar sans avoir instancié un objet de cette classe.

Nous pouvons utiliser la classe Math (**public sealed class** Math) qui contient des constantes et des fonctions mathématiques courantes :

public static const double E; // la constante **e** représente la base du logarithme népérien.
public static const double PI; // la constante **pi** représente le rapport de la circonférence d'un cercle à son diamètre.

Méthode de classe

Une méthode de classe est une méthode dont l'implémentation est la même pour tous les objets de la classe, en fait la différence avec une méthode d'instance a lieu sur la catégorie des variables sur lesquelles ces méthodes agissent.

De par leur définition les méthodes de classe ne peuvent travailler qu'avec des variables de classe, alors que les méthodes d'instances peuvent utiliser les deux catégories de variables.

Un programme correct illustrant le discours :

C#	Explications

class Exemple **{**
public static int x ;
int y ;
public void f1(**int** a)
{ x = a;
y = a;
}
public static void g1(int a)
{ x = a;
}
}
class Utilise **{**
public static void Main( ) **{**
Exemple obj = **new** Exemple( );
obj.f1(10);
System.Console.WriteLine("<f1(10)>obj.x="+ Exemple.x);
Exemple.g1(50);
System.Console.WriteLine("<g1(50)>obj.x="+ Exemple.x);
}
}
	public void f1(**int** a)
	{ x = a; *//accès à la variable de classe*
	y = a ; *//accès à la variable d'instance*
	}
	
	public static void g1(**int** a)
	{ x = a; *//accès à la variable de classe*
	y = a ; *//engendrerait un erreur de compilation : accès à une variable non static interdit !*
	}
	La méthode f1 accède à toutes les variables de la classe Exemple, la méthode g1 n'accède qu'aux variables de classe (**static** et **public**).
	
	Après exécution on obtient :
	
	<f1(10)>obj.x = 10
	<g1(50)>obj.x = 50

Résumé pratique sur les membres de classe en C#

1) - Les méthodes et les variables de classe sont **précédées obligatoirement** du mot clef **static**. Elles jouent un rôle **semblable** à celui qui est attribué aux variables et aux sous-routines globales dans un langage impératif classique.

C#	Explications
class Exemple1 **{** **int** a = 5; **static int** b = 19; **void** m1( ){...} **static void** m2( ) {...} **}**	La variable a dans **int** a = 5; est une variable d'instance. La variable b dans **static int** b = 19; est une variable de classe. La méthode m2 dans **static void** m2( ) {...} est une méthode de classe.

2) - Pour utiliser une variable **x1** ou une méthode **meth1** de la classe **Classe1**, il suffit de d'écrire **Classe1.x1** ou bien **Classe1.meth1**.

C#	Explications
class Exemple2 **{** **public static int** b = 19; **public static void** m2( ) {...} **}** **class** UtiliseExemple **{ Exemple2**.b = 53; **Exemple2**.m2( ); ... **}**	Dans la classe Exemple2, b est une variable de classe, m2 une méthode de classe. La classe UtiliseExemple fait appel à la méthode m2 directement avec le nom de la classe, il en est de même avec le champ b de la classe Exemple2

3) - Une variable de classe (précédée du mot clef **static**) est **partagée par tous les objets** de la même classe.

C#	Explications
class AppliStatic **{ public static int** x = -58 ; **public int** y = 20 ; ... **}** **class** Utilise **{** **public static void** Main(String [ ] arg) **{** AppliStatic obj1 = **new** AppliStatic( );	Dans la classe AppliStatic **x** est une variable de classe, et **y** une variable d'instance. La classe Utilise crée 3 objets (obj1,obj2,obj3) de classe AppliStatic. L'instruction obj1.**y** = 100; est un accès au champ **y** de l'instance obj1. Ce n'est que le champ **y** de cet objet qui est modifié,les champs **y** des objets obj2 et

AppliStatic obj2 = **new** AppliStatic( );
AppliStatic obj3 = **new** AppliStatic( );
obj1.y = 100;
AppliStatic.x = 101;
System.Console.WriteLine("obj1.x="+ AppliStatic.x);
System.Console.WriteLine("obj1.y="+obj1.y);
System.Console.WriteLine("obj2.x="+ AppliStatic.x);
System.Console.WriteLine("obj2.y="+obj2.y);
System.Console.WriteLine("obj3.x="+ AppliStatic.x);
System.Console.WriteLine("obj3.y="+obj3.y);
AppliStatic.x = 99;
System.Console.WriteLine("AppliStatic.x="
+ AppliStatic.x);
}
}
	obj3 restent inchangés
	
	Il n'y a qu'une seule manière d'accéder à la variable static **x**, comme pour toute variable de classe, en utilisant le nom de la classe :
	
	AppliStatic.**x** = 101;
	
	Dans ce cas cette variable **x** est modifiée globalement et donc tous les champs **x** de tous les objets obj1, obj2 et obj3 prennent la nouvelle valeur.

Au début lors de la création des 3 objets, chacun des champs x vaut -58 et chacun des champs y vaut 20, l'affichage par System.Console.WriteLine(...) donne les résultats suivants qui démontrent le partage de la variable x par tous les objets.

> ***Après exécution :***
> obj1.x = 101
> obj1.y = 100
> obj2.x = 101
> obj2.y = 20
> obj3.x = 101
> obj3.y = 20
> AppliStatic.x = 99

4) - Une méthode de classe (précédée du mot clef **static**) **ne peut utiliser que des variables de classe** (précédées du mot clef **static**) et jamais des variables d'instance.Une méthode d'instance peut accéder aux deux catégories de variables

5) - Une méthode de classe (précédée du mot clef **static**) **ne peut appeler** (invoquer) **que des méthodes de classe** (précédées du mot clef **static**).

C#	Explications
class AppliStatic **{** **public static int** x = -58 ; **public int** y = 20 ; **public void** f1(**int** a) **{** AppliStatic.x = a; y = 6 ; **}** **}** **class** Utilise **{** **static void** f2(**int** a) **{** AppliStatic.x = a;	Nous reprenons l'exemple précédent en ajoutant à la classe AppliStatic une méthode interne f1 : **public void** f1(**int** a) **{** AppliStatic.x = a; y = 6 ; **}** Cette méthode accède à la variable de classe comme un champ d'objet. Nous rajoutons à la classe Utilise, un méthode static (méthode de classe) notée f2: **static void** f2(**int** a) **{** AppliStatic.x = a; **}** Cette méthode accède elle aussi à la variable de

`}`
 	
`public static void Main( ) {`
`AppliStatic obj1 = new AppliStatic( );`
`AppliStatic obj2 = new AppliStatic( );`
`AppliStatic obj3 = new AppliStatic( );`
`obj1.y = 100;`
 	
`obj1.x = 101;`
`AppliStatic.x = 99;`
`f2(101);`
`obj1.f1(102);`
`}`
`}`
	classe parce qu c'est une méthode static.
	
	Nous avons donc **trois** manières d'accéder à la variable static **x**, :
	
	soit directement comme une variable de classe proprement dite : `AppliStatic.x = 99;`
	
	soit indirectement par une méthode d'instance sur son champ : `obj1.f1(102);`
	
	soit indirectement par une méthode **static** (de classe) : `f2(101);`

Comme la méthode Main est **static,** elle peut invoquer la méthode f2 qui est aussi **static.**

Au paragraphe précédent, nous avons indiqué que C# ne connaissait pas la notion de variable globale stricto sensu, mais en fait une variable **static peut jouer le rôle d'un variable globale pour un ensemble d'objets** instanciés à partir de la même classe.

Attention :

> Contrairement à Java qui autorise l'accès à une variable **static** à partir d'un nom d'objet à la place du nom de la classe de définition, le compilateur C# impose une cohérence dans les déclarations, en refusant cette possibilité.

Depuis la version 2.0, une classe peut être **static**, dans ce cas elle correspond à un module de bibliothèque de membres (méthodes et variables) obligatoirement tous **static**.

```
static class AppliStatic
{
  public static int x = -58 ;
  public static int y = 20 ;

 public static void f1(int a)
 {  x = a;
    y = 6 ;
 }
}
```

Polymorphisme d'objet en

Plan général:

Rappel des notions de base

Polymorphisme d'objet en C# : définitions

1.1 Instanciation et utilisation dans le même type

1.2 Polymorphisme d'objet implicite

1.3 Polymorphisme d'objet explicite par transtypage

1.4 Utilisation pratique du polymorphisme d'objet

1.5 Instanciation dans un type ascendant impossible

Le polymorphisme en C#

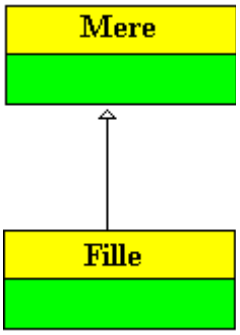
Rappel utile sur les notions de bases

Il existe un concept essentiel en POO désignant la capacité d'une hiérarchie de classes à fournir différentes implémentations de méthodes portant le même nom et par corollaire la capacité qu'ont des objets enfants de modifier les comportements hérités de leur parents. Ce concept d'adaptation à différentes "situations" se dénomme le **polymorphisme** qui peut être implémenté de différentes manières.

Polymorphisme d'objet C'est une interchangeabilité entre variables d'objets de classes de la même hiérarchie sous certaines conditions, que dénommons le polymorphisme d'objet.
Polymorphisme par héritage de méthode Lorsqu'une classe enfant hérite d'une classe mère, des méthodes supplémentaires nouvelles peuvent être implémentées dans la classe enfant mais aussi des méthodes des parents peuvent être substituées pour obtenir des implémentations différentes.
Polymorphisme par héritage de classes abstraites Une classe abstraite est une classe qui ne peut pas s'instancier elle-même ; elle doit être héritée. Certains membres de la classe peuvent ne pas être implémentés, et c'est à la classe qui hérite de fournir cette implémentation.
Polymorphisme par implémentation d'interfaces Une interface décrit la signature complète des membres qu'une classe doit implémenter, mais elle laisse l'implémentation de tous ces membres à la charge de la classe d'implémentation de l'interface.

Polymorphisme d'objet en C#

Soit une classe **Mere** et une classe **Fille** héritant de la classe **Mere** :



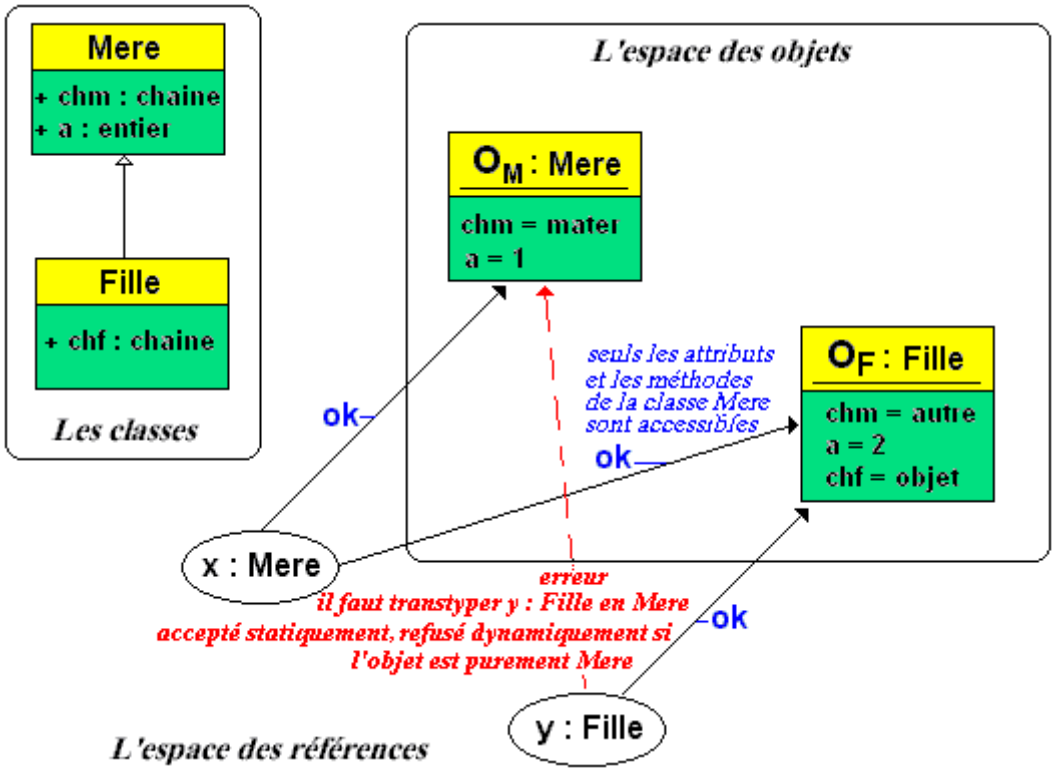
Les objets peuvent avoir des comportements polymorphes (s'adapter et se comporter différement selon leur utilisation) licites et des comportements polymorphes dangereux selon les langages.

Dans un langage dont le modèle objet est la référence (un objet est un couple : référence, bloc mémoire) comme **C#,** il y a découplage entre les actions statiques du compilateur et les actions dynamiques du système d'exécution, **le compilateur protège statiquement des actions dynamiques sur les objets** une fois créés. C'est la déclaration et l'utilisation des variables de références qui autorise ou non les actions licites grâce à la compilation.

Supposons que nous ayons déclaré deux variables de référence, l'une de classe **Mere**, l'autre de classe **Fille**, une question qui se pose est la suivante : au cours du programme quel genre d'affectation et d'instanciation est-on autorisé à effectuer sur chacune de ces variables dans un programme C#.

En C# :
public class Mere {
.....
}
public class Fille : **Mere** {
.....
}
	L'héritage permet une variabilité entre variables d'objets de classes de la même hiérarchie, c'est cette variabilité que dénommons le **polymorphisme d'objet**.

Nous envisageons toutes les situations possibles et les évaluons, les exemples explicatifs sont écrits en C# (lorsqu'il y a discordance avec java ou Delphi autres langages, celle-ci est mentionnée explicitement), il existe 3 possibilités différentes illustrées par le schéma ci-dessous.

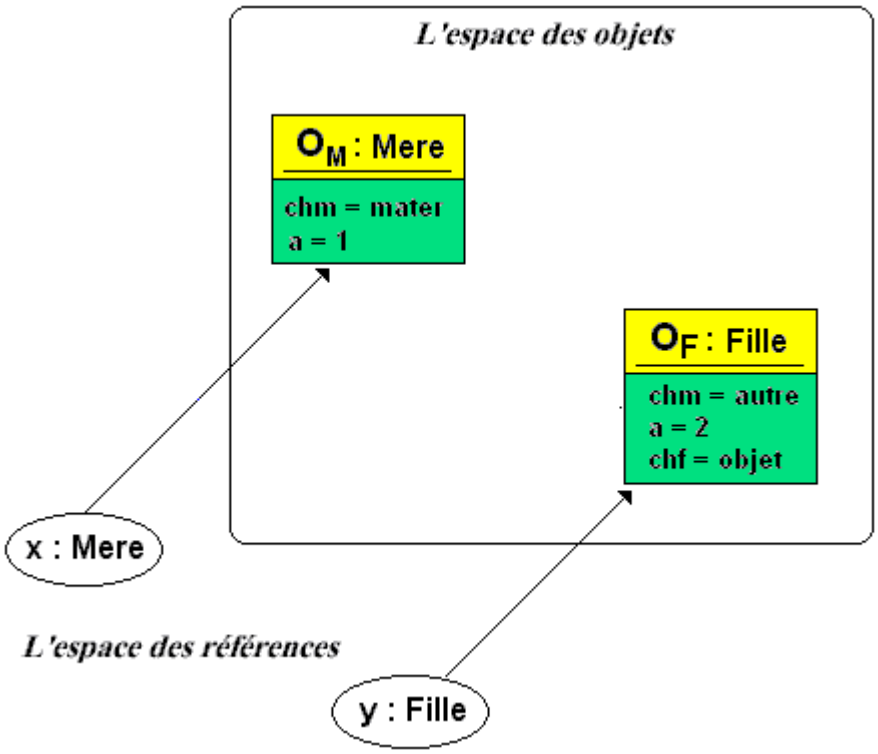


L'instanciation et l'utilisation de références dans le même type
L'affectation de références : polymorphisme implicite
L'affectation de références : polymorphisme par transtypage d'objet

La dernière de ces possibilités pose un problème d'exécution lorsqu'elle mal employée !

1.1 instanciation dans le type initial et utilisation dans le même type

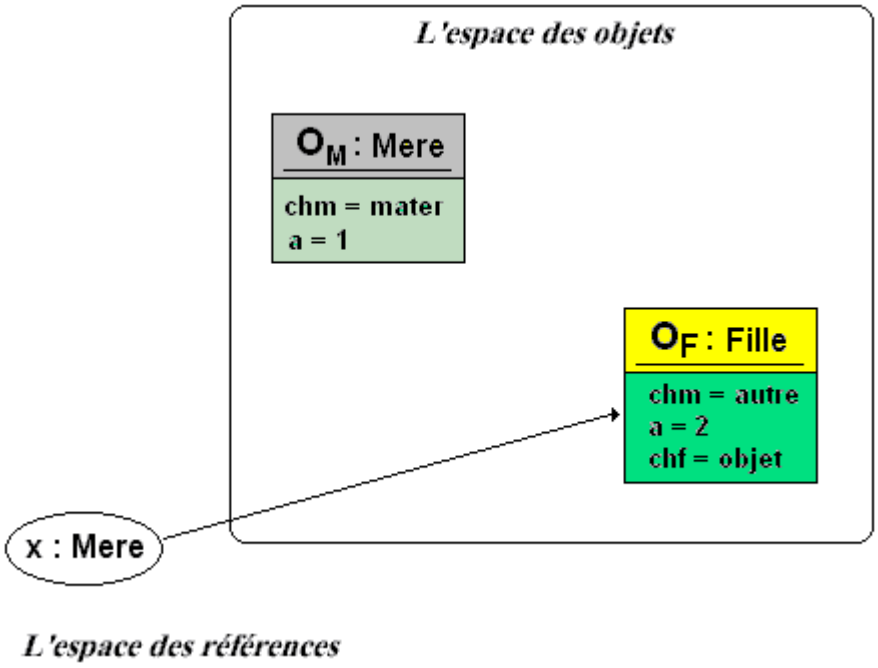
Il s'agit ici d'une utilisation la plus classique qui soit, dans laquelle une variable de référence d'objet est utilisée dans son type de définition initial (valable dans tous les LOO)



En C# :

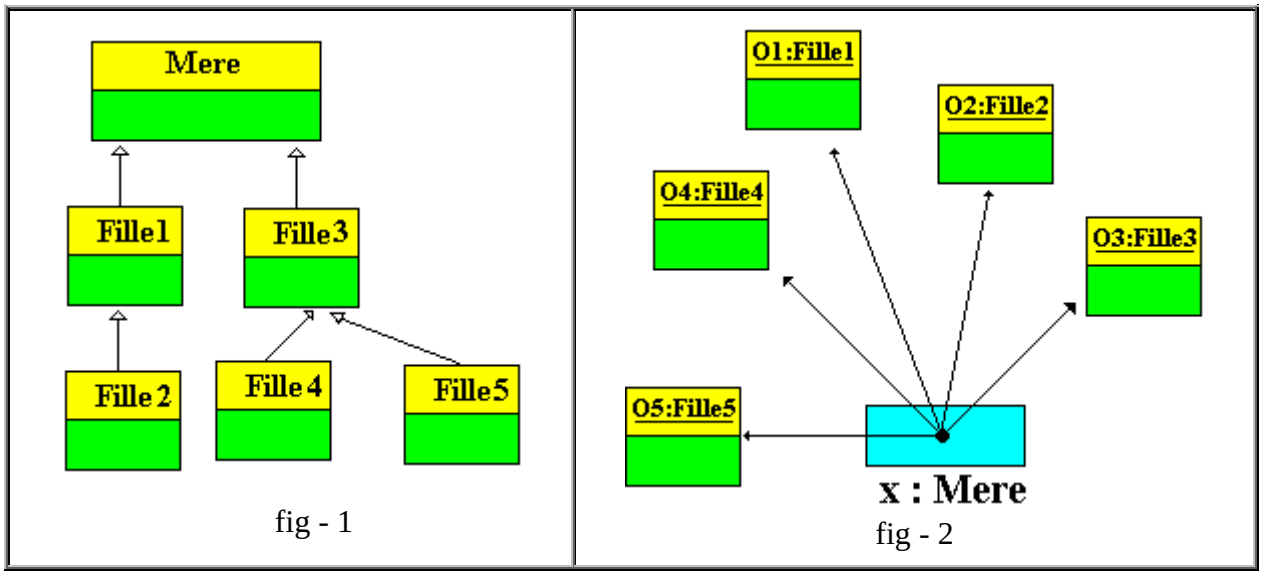
```
Mere x , u ;
Fille y , w ;
.....
x  = new Mere(  ) ; // instanciation dans le type initial
u = x ; // affectation de références du même type

y = new Fille(  ) ;  // instanciation dans le type initial
v = y ; // affectation de références du même type
```

1.2 Polymorphisme d'objet implicite




```
En C# :

Mere x ;
Fille  ObjF = new  Fille( )   ;
x = ObjF; // affectation de références du type descendant implicite
```

Nous pouvons en effet dire que **x** peut se référer implicitement à tout objet de classe **Mere** ou de **toute classe héritant** de la classe **Mere**.



Dans la figure fig-1 ci-dessus, une hiérarchie de classes decendant toutes de la classe **Mere**, dans fig-2 ci-contre le schéma montre une référence de type **Mere** qui peut '**pointer**' vers n'importe quel objet de classe descendante (polymorphisme d'objet).

D'une façon générale vous pourrez toujours écrire des affectations entre deux références d'objets :

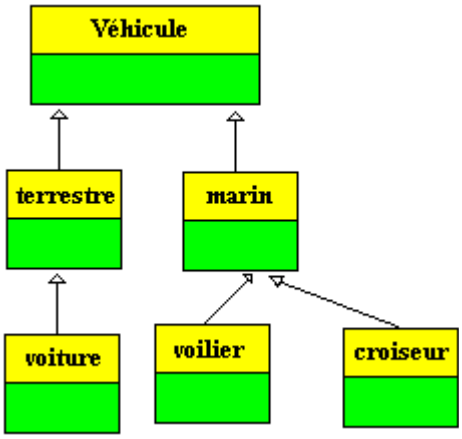
En C# :

```
Classe1  x ;
Classe2  y ;
..........
x = y ;
```

si et seulement si **Classe2** est une classe descendante de **Classe1.**

Exemple pratique tiré du schéma précédent

1°) Le polymorphisme d'objet est typiquement fait pour représenter des situations pratiques figurées ci-dessous :



Une hiérarchie de classe de véhicules descendant toutes de la classe mère Vehicule, on peut énoncer le fait suivant :
Un véhicule peut être de plusieurs sortes : soit un croiseur, soit une voiture, soit un véhicule terrestre etc...

Traduit en termes informatiques, si l'on déclare une référence de type véhicule **(vehicule** x) elle pourra pointer vers n'importe quel objet d'une des classe filles de la classe vehicule.

En C# :

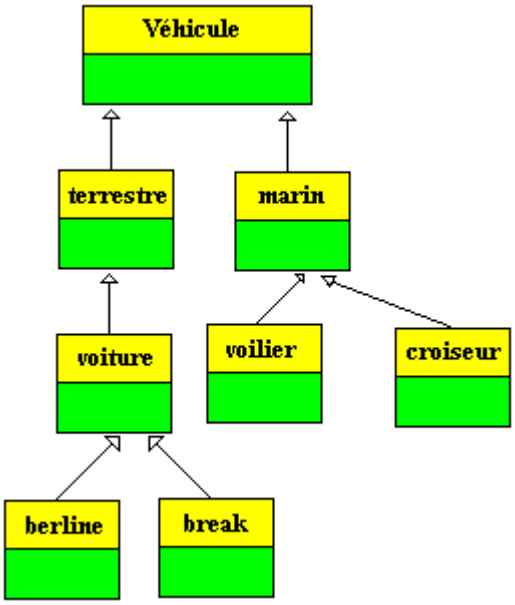
```
public class Vehicule {
  ..........
}
public class terrestre : Vehicule{
 ..........
}
public class voiture : terrestre {
  ..........
}
```

```
public class marin : Vehicule {
 ..........
}

public class voilier : marin {
  ..........
}

public class croiseur : marin {
 ..........
}
```

Mettons en oeuvre la définition du polymorphisme implicite :

Polymorphisme implicite = création d'objet de classe descendante référencé par une variable parent

Ajoutons 2 classes à la hiérarchie des véhicules :



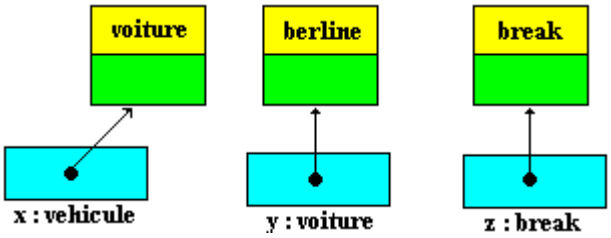
Partons de la situation pratique suivante :

- on crée un **véhicule** du type **voiture** ,
- on crée une **voiture** de type **berline** ,
- enfin on crée un **break** de type **break**

Traduit en termes informatiques : nous déclarons 3 références **x**, **y** et **z** de type **vehicule**, **voiture** et **break** et nous créons 3 objets de classe **voiture**, **berline** et **break**.

Comme il est possible de créer directement un objet de classe descendante à partir d'une référence de classe mère, nous proposons les instanciations suivantes :

- on crée une **voiture** référencée par la variable de classe **vehicule**,
- on crée une **berline** référencée par la variable de classe **voiture**,
- enfin on crée un **break** référencé par la variable de classe **break**.



En C# :
 	
public class berline : voiture {
.....
}
 	
public class break : voiture {
.....
}
	public class Fabriquer
	{
	Vehicule x = **new** voiture ( ) ;
	voiture y = **new** berline ( ) ;
	break z = **new** break ( );
	.....
	}

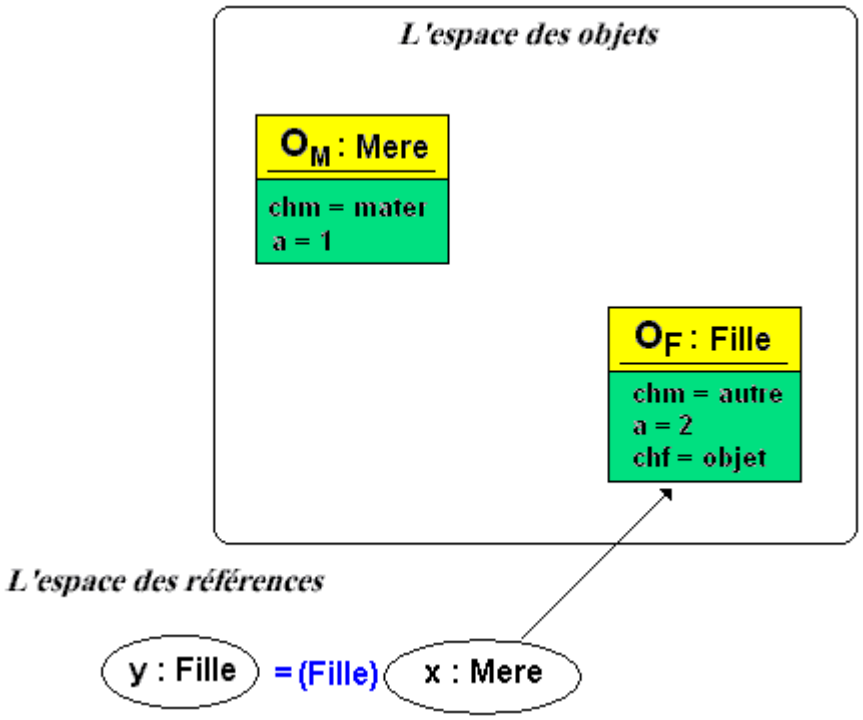
1.3 Polymorphisme d'objet explicite par transtypage

La situation informatique est la suivante :

- on déclare une variable **x** de type Mere,
- on déclare une variable **y** de type Fille héritant de Mere,
- on instancie la variable x dans le type descendant Fille (polymorphisme implicite).

Il est alors possible de faire "pointer" la variable y (de type Fille) vers l'objet (de type Fille) auquel se réfère x en effectuant une affectation de références :

y = x ne sera pas acceptée directement car statiquement les variables x et y ne sont pas du même type, il faut indiquer au compilateur que l'on souhaite temporairement changer le type de la variable x afin de pouvoir effectuer l'affectation.
Cette opération de changement temporaire, se dénomme le **transtypage** (notée en C# : y = (Fille)x) :



En C# :

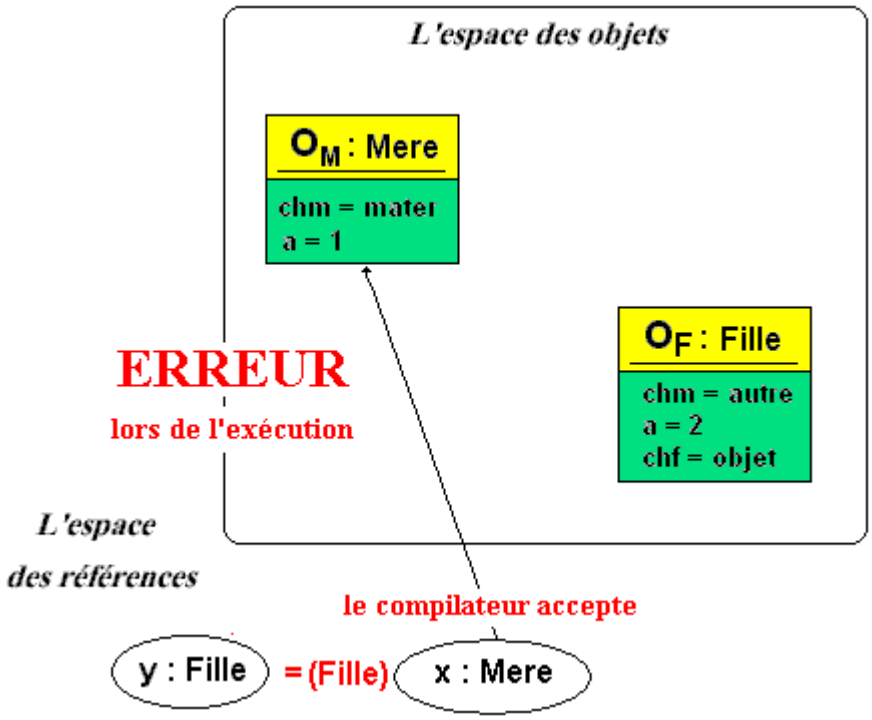
```
Mere x ;
Fille y  ;
Fille  ObjF = new  Fille( )   ;
x = ObjF ; // x pointe vers un objet de type Fille
y = (Fille) x ; // transtypage et affectation de références du type ascendant explicite compatible dynamiquement.
```

Attention

- La validité du transtypage n'est pas vérifiée statiquement par le compilateur, donc si votre variable de référence pointe vers un objet qui n'a pas la même nature que l'opérateur de transtypage, c'est lors de l'exécution qu'il y aura production d'un message d'erreur indiquant le transtypage impossible.

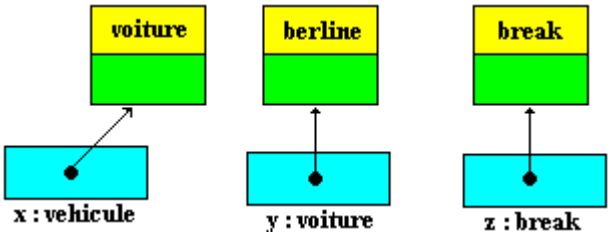
- Il est donc impératif de tester l'appartenance à la bonne classe de l'objet à transtyper avant de le transtyper, les langages C#, Delphi et Java disposent d'un opérateur permettant de tester cette appartenance ou plutôt l'appartenance à une hiérarchie de classes (opérateur **is** en C#).

- L'opérateur **as** est un opérateur de transtypage de référence d'objet semblable à l'opérateur ( ). L'opérateur **as** fournit la valeur **null** en cas d'échec de conversion alors que l'opérateur ( ) lève une exception.



```
En C# :
Mere x   ;
Fille y   ;
x = new  Mere( ) ; // instanciation dans le type initial
{ affectation de références du type ascendant explicite mais dangereuse si x est uniquement Mere : }
 y = (Fille)x ;  <--- erreur lors de l'exécution ici
{affectation acceptée statiquement mais refusée dynamiquement, car x pointe vers un objet de type Mere }
```

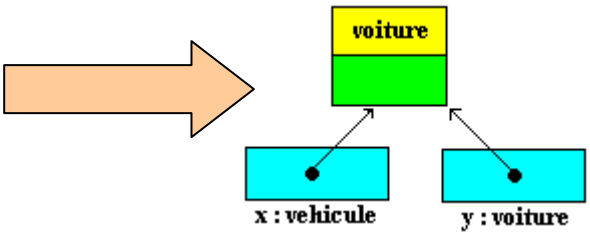
En reprenant l'exemple pratique précédant de la hiérarchie des véhicules :



Puisque x pointe vers un objet de type voiture toute variable de référence acceptera de pointer vers cet objet, en particulier la variable voiture après transtypage de la référence de x.

En C# l'affectation s'écrirait par application de l'opérateur de transtypage :

y = (voiture) x;



Pour pallier à cet inconvénient de programmation pouvant lever des exceptions lors de l'exécution, C# offre au développeur la possibilité de tester l'appartenance d'un objet référencé par une variable quelconque à une classe ou plutôt une hiérarchie de classe ; en C# cet opérateur se dénote **is** :

L'opérateur "is" de C# est identique à celui de Delphi :

L'opérateur **is**, qui effectue une vérification de type dynamique, est utilisé pour vérifier quelle est effectivement la classe d'un objet à l'exécution.

L'expression : objet **is** classeT

renvoie True si objet est une instance de la classe désignée par **classeT** ou de l'**un de ses descendants**, et False sinon. Si objet a la valeur nil, le résultat est False.

```
En C# :
Mere x   ;
Fille y   ;
x = new  Mere( ) ; // instanciation dans le type initial
if ( x is Fille) // test d'appartenance de l'objet référencé par x à la bonne classe
   y = (Fille)x ;
```

1.4 Utilisation pratique du polymorphisme d'objet

Le polymorphisme d'objet associé au transtypage est très utile dans les paramètres des méthodes.

Lorsque vous déclarez une méthode meth avec un paramètre formel **x** de type ClasseT :

```
void  meth ( ClasseT x );
{
    ........
}
```

Vous pouvez utiliser lors de l'appel de la méthode meth n'importe quel paramètre effectif de ClasseT ou bien d'une quelconque classe descendant de ClasseT et ensuite à l'intérieur de la procédure vous transtypez le paramètre. Cet aspect est utilisé en particulier en C# lors de la création de gestionnaires d'événements communs à plusieurs composants :

```
private void meth1(object sender, System.EventArgs e) {
  if (sender is System.Windows.Forms.TextBox)
          (sender as TextBox).Text="Fin";
 else if (sender is System.Windows.Forms.Label)
            (sender as Label).Text="ok";

 // ou encore :

 if (sender is System.Windows.Forms.TextBox)
    ( (TextBox)sender ).Text="Fin";

 else if (sender is System.Windows.Forms.Label)
            ( (Label)sender ).Text="ok";
}
```

Autre exemple avec une méthode meth2 personnelle sur la hiérarchie des véhicules :

```
private void  meth2 ( vehicule Sender );
{
    if (Sender is voiture)
      ((voiture)Sender).  ....... ;
   else if (Sender is voilier)
     ((voilier)Sender).  ....... ;
  ............
}
```

instanciation dans un type ascendant (*impossible en C#*)

- Il s'agit ici d'une utilisation non licite qui n'est pas commune à tous les langages LOO.
- Le compilateur **C#** comme le compilateur **Java**, **refuse** ce type de création d'objet, les compilateurs C++ et **Delphi** **acceptent** ce genre d'instanciation en laissant au programmeur le soin de se débrouiller avec les problèmes de cohérence lorsqu'ils apparaîtront.

Polymorphisme de méthode en

Plan général:

1. Le polymophisme de méthodes en C#

Rappel des notions de base

1.1 Surcharge et redéfinition en C#
1.2 Liaison statique et masquage en C#
1.3 Liaison dynamique en C#
1.4 Comment opère le compilateur

Résumé pratique

2. Accès à la super classe en C#

2.1 Le mot clef base
2.2 Initialiseur de constructeur this et base
2.3 Comparaison C#, Delphi et Java sur un exemple
2.4 Traitement d'un exercice complet
2.5 Destructeur-finaliseur

1. Le polymorphisme de méthode en C#

Nous avons vu au chapitre précédent le polymorphisme d'objet, les méthodes peuvent être elles aussi polymorphes. Nous avons vu comment Delphi mettait en oeuvre le polymorphisme d'objet et de méthode, nous voyons ici comment C# hérite une bonne part de Delphi pour ses comportements et de la souplesse de Java pour l'écriture.

Rappel de base

Polymorphisme par héritage de méthode

Lorsqu'une classe enfant hérite d'une classe mère, des méthodes supplémentaires **nouvelles** peuvent être implémentées dans la classe enfant, mais aussi des méthodes des **parents substituées** pour obtenir des implémentations différentes.

L'objectif visé en terme de qualitié du logiciel est la **réutilisabilité** en particulier lorsque l'on réalise une même opération sur des éléments différents :

opération = **ouvrir** ( )
ouvrir une fenêtre de texte, **ouvrir** un fichier, **ouvrir** une image etc ...

Surcharge et redéfinition avec C#

En informatique ce vocable s'applique aux méthodes selon leur degré d'adaptabilité, nous distinguons alors deux dénominations :

- le polymorphisme statique ou la **surcharge** de méthode
- le polymorphisme dynamique ou la **redéfinition** de méthode ou encore la surcharge héritée.

1.1 Surcharge

La surcharge de méthode (*polymorphisme statique de méthode*) est une fonctionnalité classique des langages très évolués et en particulier des langages orientés objet dont C# fait partie; elle consiste dans le fait qu'une classe peut disposer de **plusieurs méthodes ayant le même nom**, mais avec des paramètres formels différents ou éventuellement un type de retour différent. On dit alors que ces méthodes n'ont pas la même signature

On rappelle que la **signature** d'une méthode est formée par l'**en-tête** de la méthode avec ses **paramètres formels** et leur **type**.

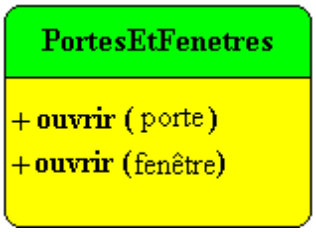
Nous avons déjà utilisé cette fonctionnalité précédement dans le paragraphe sur les constructeurs, où la classe **Un** disposait de quatre constructeurs surchargés (quatre **signatures** différentes du constructeur) :

```
class Un
 {
  int a;
   public Un ( )
   { a = 100;   }

   public Un (int b )
```

```
  { a = b;    }

  public Un (float b )
  { a = (int)b;    }

  public Un (int x , float y ) : this(y)
  { a += 100;    }
}
```

Mais cette surcharge est possible aussi pour n'importe quelle méthode de la classe autre que le constructeur.

Ci-contre deux exemplaires surchargés de la méthode ouvrir dans la classe **PortesEtFenetres :**



Le compilateur n'éprouve aucune difficulté lorsqu'il rencontre un appel à l'une des versions surchargée d'une méthode, il cherche dans la déclaration de toutes les surcharges celle dont la **signature** (la déclaration des paramètres formels) coïncide avec les paramètres effectifs de l'appel.

Remarque :

Le polymorphisme statique (ou **surcharge**) de C# est syntaxiquement semblable à celui de Java.

Programme C# exécutable	Explications

class Un
{ public int a;
public Un (**int** b)
{ a = b; }
public void f ( )
{ a *=10; **}**
public void f (**int** x)
{ a +=10*x; **}**
public int f (**int** x, **char** y)
{ a = x+(**int**)y;
return a; **}**
}
class AppliSurcharge **{**
public static void Main(String [ ] arg) **{**
Un obj = **new** Un(15);
System.Console.WriteLine("<création> a ="+obj.a);
obj.f( );
System.Console.WriteLine("<obj.f()> a ="+obj.a);
obj.f(2);
System.Console.WriteLine("<obj.f()> a ="+obj.a);
obj.f(50,'a');
System.Console.WriteLine("<obj.f()> a ="+obj.a);
}
}
	La méthode **f** de la classe **Un** est surchargée trois fois :
	
	public void f ( )
	{ a *=10; **}**
	
	public void f (**int** x)
	{ a +=10*x; **}**
	
	public int f (**int** x, **char** y)
	{ a = x+(int)y;
	return a;
	}
	
	La méthode f de la classe Un peut donc être appelée par un objet instancié de cette classe sous l'une quelconque des trois formes :
	
	obj.f( ); pas de paramètre => choix : **void** f ( )
	
	obj.f(2); paramètre **int** => choix : **void** f (**int** x)
	
	obj.f(50,'a'); deux paramètres, un int un char => choix : **int** f (**int** x, **char** y)

Comparaison Delphi - C# sur la surcharge :

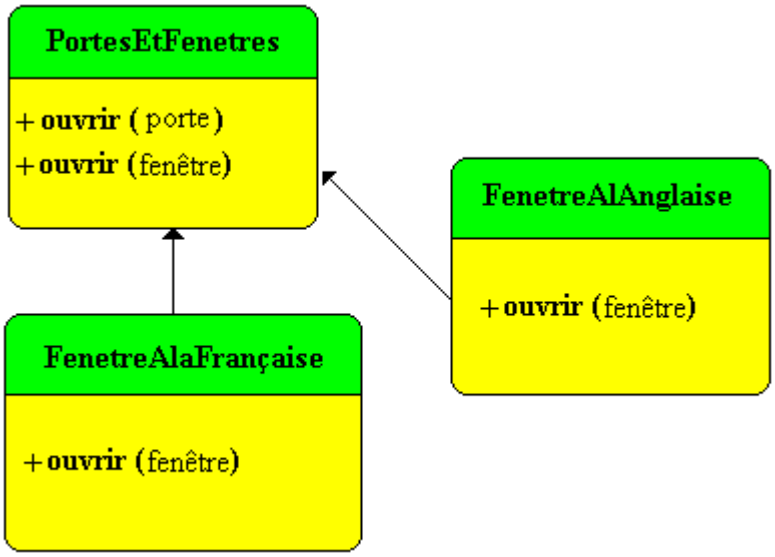
Delphi	C#

Un = **class**
a : integer;
public
constructor methode(b : integer);
procedure f;**overload**;
procedure f(x:integer);**overload**;
function f(x:integer;y:char):integer;**overload**;
end;
 	
implementation
 	
constructor Un.methode(b : integer); **begin**
a:=b
end;
procedure Un.f; **begin**
a:=a*10;
end;
procedure Un.f(x:integer); **begin**
a:=a+10*x
end;
function Un.f(x:integer;y:char):integer; **begin**
a:=x+ord(y);
result:= a
end;
 	
procedure Main;
var obj:Un;
begin
obj:=Un.methode(15);
obj.f;
Memo1.Lines.Add('obj.f='+inttostr(obj.a));
obj.f(2);
Memo1.Lines.Add('obj.f(2)='+inttostr(obj.a));
obj.f(50,'a');
Memo1.Lines.Add('obj.f(50,"a")='+inttostr(obj.a));
end;
	class Un
	{
	public int a;
	public Un (**int** b)
	{ a = b;
	}
	
	public void f ( )
	{ a *=10;
	}
	public void f (**int** x)
	{ a +=10*x;
	}
	public int f (**int** x, **char** y)
	{ a = x+(**int**)y;
	return a;
	}
	}
	
	class AppliSurcharge
	{
	public static void Main(String [ ] arg) **{**
	Un obj = **new** Un(15);
	System.Console.WriteLine("<création> a ="+obj.a);
	obj.f( );
	System.Console.WriteLine("<obj.f()> a ="+obj.a);
	obj.f(2);
	System.Console.WriteLine("<obj.f()> a ="+obj.a);
	obj.f(50,'a');
	System.Console.WriteLine("<obj.f()> a ="+obj.a);
	}
	}

Redéfinition

La redéfinition de méthode (ou polymorphisme dynamique) est spécifique aux langages orientés objet. Elle est mise en oeuvre lors de l'héritage d'une classe mère vers une classe fille dans le cas d'une méthode ayant la même signature dans les deux classes. Dans ce cas les actions dûes à l'appel de la méthode, dépendent du code inhérent à chaque version de la méthode (celle de la classe mère, ou bien celle de la classe fille).

Dans l'exemple ci-dessous, nous supposons que dans la classe **PortesEtFenetres** la méthode ouvrir(fenetre) explique le mode opératoire général d'ouverture d'une fenêtre, il est clair que dans les deux classes descendantes l'on doit "redéfinir" le mode opératoire selon que l'on est en présence d'une fenêtre à la française, ou d'une fenêtre à l'anglaise :



Redéfinition et répartition des méthodes en C#

La redéfinition de méthode peut être selon les langages :

- **précoce**
  et/ou
- **tardive**

Ces deux actions sont différentes selon que le compilateur du langage met en place la laison du code de la méthode immédiatement lors de la compilation (**liaison statique** ou **précoce**) ou bien lorsque le code est lié lors de l'exécution (**laison dynamique** ou **tardive**). Ce phénomène se dénomme la répartition des méthodes.

Le terme de répartition fait référence à la façon dont un programme **détermine où il doit rechercher** une méthode lorsqu'il rencontre un appel à cette méthode.

Le code qui appelle une méthode ressemble à un appel classique de méthode. Mais les classes ont des façons différentes de répartir les méthodes.

Le langage **C#** supporte d'une manière **identique à Delphi**, ces deux modes de liaison du code, la **liaison statique** étant comme en Delphi le mode **par défaut**.

Le développeur Java sera plus décontenancé sur ce sujet, car la liaison statique en Java n'existe que pour les methodes de classe **static**, de plus **la liaison du code par défaut est dynamique en Java**.

Donc en C# comme en Delphi , des mots clefs comme **virtual** et **override** sont nécessaires pour la redéfinition de méthode, ils sont utilisés strictement de la même manière qu'en Delphi.

1.2 Liaison statique et masquage en C#

Toute méthode C# qui n'est précédée d'aucun des deux qualificateurs **virtual** ou **override** est à liaison **statique**.

Le compilateur détermine l'adresse exacte de la méthode et lie la méthode au moment de la compilation.

L'avantage principal des méthodes statiques est que leur répartition est très rapide. Comme le compilateur peut déterminer l'adresse exacte de la méthode, il la lie directement (les méthodes virtuelles, au contraire, utilisent un moyen indirect pour récupérer l'adresse des méthodes à l'exécution, moyen qui nécessite plus de temps).

Une méthode statique ne change pas lorsqu'elle est transmise en héritage à une autre classe. Si vous déclarez une classe qui inclut une méthode statique, puis en dérivez une nouvelle classe, la classe dérivée partage exactement la même méthode située à la même adresse. Cela signifie qu'il est impossible de redéfinir les méthodes statiques; une méthode statique fait toujours exactement la même chose, quelque soit la classe dans laquelle elle est appelée.

Si vous déclarez dans une classe dérivée une méthode ayant le même nom qu'une méthode statique de la classe ancêtre, la nouvelle méthode remplace simplement (on dit aussi masque) la méthode héritée dans la classe dérivée.

Comparaison masquage en Delphi et C# :

Delphi	C#
type ClasseMere = **class** x : integer; **procedure** f (a:integer); **end**; ClasseFille = **class** (ClasseMere) y : integer; **procedure** f (a:integer);*//masquage* **end**; **implementation** **procedure** ClasseMere.f (a:integer); **begin...** **end**; **procedure** ClasseFille.f (a:integer); **begin...** **end**;	**public class** ClasseMere **{** **public int** x = 10; **public void** f (**int** a) **{** x +=a; **}** **}** **public class** ClasseFille **:** ClasseMere **{** **int** y = 20; **public void** f (**int** a) *//masquage* **{** x +=a*10+y; **}** **}**

Remarque importante :

L'expérience montre que les étudiants comprennent immédiatement le masquage lorsque le polymorphisme d'objet n'est pas présent. Ci-dessous un exemple de classe UtiliseMereFille qui instancie et utilise dans le même type un objet de classe ClasseMere et un objet de classe ClasseFille :

```
public class ClasseMere
{
 public int x = 1;
 public  void meth ( int a)
 {
  x += a ;
 }
}
class ClasseFille : ClasseMere
{
 public  void meth ( int a) //masquage
 {
  x += a*100 ;
 }
}

public class UtiliseMereFille
{
 public  static void Main (string [ ] args)
 {
  ClasseMere M ;
  ClasseFille F ;
  M = new ClasseMere ( ) ;
  F = new ClasseFille ( ) ;
  M.meth (10) ;
  F.meth (10) ;
 }
}
```

- Lors de la compilation de l'instruction M.meth(10), c'est le code de la méthode meth de la classe ClasseMere qui est lié avec comme paramètre par valeur 10; ce qui donnera la valeur 11 au champ x de l'objet M.

- Lors de la compilation de l'instruction F.meth(10), c'est le code de la méthode meth de la classe ClasseFille qui masque celui de la classe parent et qui est donc lié avec comme paramètre par valeur 10; ce qui donnera la valeur 101 au champ x de l'objet F.

Pour bien comprendre toute la portée du masquage statique et les risques de mauvaises interprétations, il faut étudier le même exemple légèrement modifié en incluant le cas du polymorphisme d'objet, plus précisément le polymorphisme d'objet implicite.

Dans l'exemple précédent nous instancions la variable ClasseMere M en un objet de classe ClasseFille (*polymorphisme implicite d'objet*) soient les instructions

ClasseMere M ;
M = **new** ClasseFille ( ) ;

Une erreur courante est de croire que dans ces conditions, dans l'instruction **M.meth(10)** c'est la méthode meth(**int** a) de la classe ClasseFille (en particulier si l'on ne connaît que Java qui ne pratique pas le masquage) :

```
public class ClasseMere
{
 int x = 1;
 public  void meth ( int a)
 {
  x += a ;
 }
}
class ClasseFille : ClasseMere
{
 public  void meth ( int a) //masquage
 {
  x += a*100 ;
 }
}

public class UtiliseMereFille
{
 public  static void Main (string [ ] args)
 {
  ClasseMere M ;
  ClasseFille F ;
  M = new ClasseFille( ) ;
  F = new ClasseFille ( ) ;
  M.meth (10) ;
  F.meth (10) ;
 }
}
```

ERREUR

Que fait alors le compilateur C# dans ce cas ? : il réalise une liaison statique :

- Lors de la compilation de l'instruction **M.meth(10)**, c'est le code de la méthode meth(**int** a) de la classe ClasseMere qui est lié, car la référence M a été déclarée de type ClasseMere et peu importe dans quelle classe elle a été instanciée (avec comme paramètre par valeur 10; ce qui donnera la valeur 11 au champ x de l'objet M).

- Lors de la compilation de l'instruction F.meth(10), c'est le code de la méthode meth de la classe ClasseFille comme dans l'exemple précédent (avec comme paramètre par valeur 10; ce qui donnera la valeur 101 au champ x de l'objet F).

Voici la bonne configuration de laison effectuée lors de la compilation :

```
public class ClasseMere
{
 int x = 1;
 public  void meth ( int a)
 {
  x += a ;
 }
}
class ClasseFille : ClasseMere
{
 public  void meth ( int a) //masquage
 {
  x += a*100 ;
 }
}

public class UtiliseMereFille
{
 public  static void Main (string [ ] args)
 {
  ClasseMere M ;
  ClasseFille F ;
  M = new ClasseFille ( ) ;
  F = new ClasseFille ( ) ;
  M.meth (10) ;
  F.meth (10) ;
 }
}
```

Afin que le programmeur soit bien conscient d'un effet de masquage d'une méthode héritée par une méthode locale, le compilateur C# envoie, comme le compilateur Delphi, un message d'avertissement indiquant une possibilité de **manque de cohérence sémantique** ou un **masquage**.

S'il s'agit d'un masquage voulu, le petit plus apporté par le langage C# est la proposition que vous fait le compilateur de l'utilisation optionnelle du mot clef **new** qualifiant la nouvelle méthode masquant la méthode parent. Cette écriture améliore la lisibilité du programme et permet de se rendre compte que l'on travaille avec une liaison statique. Ci-dessous deux écritures équivalentes du masquage de la méthode meth de la classe ClasseMere :

masquage **C#**	masquage **C#** avec **new**
public class ClasseMere **{** **int** x = 10; **public void** meth (**int** a) *//liaison statique* **{** x +=a; **}** **}** **public class** ClasseFille **:** ClasseMere **{** **public void** meth (**int** a) *//masquage* **{** x +=a*10+y; **}** **}**	**public class** ClasseMere **{** **int** x = 10; **public void** meth (**int** a) *//liaison statique* **{** x +=a; **}** **}** **public class** ClasseFille **:** ClasseMere **{** **public new void** meth (**int** a) *//masquage* **{** x +=a*10+y; **}** **}**

L'exemple ci-dessous récapitule les notions de masquage et de surcharge en C# :

```
public class ClasseMere {
  int x = 1;
  public  void meth1 ( int a) {  x += a ; }
  public  void meth1 ( int a , int b) {  x += a*b ; }
}

public class ClasseFille : ClasseMere {
  public  new void meth1 ( int a) {  x += a*100 ; }
  public  void meth1 ( int a , int b , int c) {  x += a*b*c ; }
}


public class UtiliseMereFille {
 public  static void Main (string [ ] args) {
  ClasseMere M ;
  ClasseFille F ;
  M = new ClasseFille ( ) ;
  F = new ClasseFille ( ) ;
  M.meth1 (10) ; <--- meth1(int a) de ClasseMere
  M.meth1 (10,5) ; <--- meth1(int a, int b) de ClasseMere
  M.meth1 (10,5,2) ; <--- erreur! n'existe pas dans ClasseMere .
  F.meth1 (10) ; <--- meth1(int a) de ClasseFille
  F.meth1 (10,5) ; <--- meth1(int a, int b) de ClasseFille
  F.meth1 (10,5,2) ; <--- meth1(int a, int b, int c) de ClasseFille
 }
}
```

1.3 Liaison dynamique (ou redéfinition) en C#

Dans l'exemple ci-dessous la classe ClasseFille qui hérite de la classe ClasseMere, redéfini la méthode **f** de sa classe mère :

Comparaison redéfinition Delphi et C# :

Delphi	C#

type
ClasseMere = **class**
x : integer;
procedure f (a:integer);**virtual**;*//autorisation*
procedure g(a,b:integer);
end;
 	
ClasseFille = **class** (ClasseMere)
y : integer;
procedure f (a:integer);**override**;*//redéfinition*
procedure g1(a,b:integer);
end;
 	
implementation
 	
procedure ClasseMere.f (a:integer); **begin...**
end;
procedure ClasseMere.g(a,b:integer); **begin...**
	class ClasseMere
	{
	public int x = 10;
	
	public virtual void f (**int** a)
	{ x +=a; }
	void g (**int** a, **int** b)
	{ x +=a*b; }
	}
	
	class ClasseFille **:** ClasseMere
	{
	int y = 20;
	public override void f (**int** a) *//redéfinition*
	{ x +=a; }
	void g1 (**int** a, **int** b) *//nouvelle méthode*
	{ ..... }
	}

end; **procedure** ClasseFille.f (a:integer); **begin...** **end**; **procedure** ClasseFille.g1(a,b:integer); **begin...** **end**;	

Comme delphi, C# peut combiner la surcharge et la redéfinition sur une même méthode, c'est pourquoi nous pouvons parler de surcharge héritée :

C#
class ClasseMere **{** **public int** x = 10; **public virtual void** f (**int** a) **{** x +=a; **}** **public virtual void** g (**int** a, **int** b) **{** x +=a*b; **}** **}** **class** ClasseFille **:** ClasseMere **{** **int** y = 20; **public override void** f (**int** a) *//redéfinition* **{** x +=a; **}** **public virtual void** g (**char** b) *//surcharge de g* **{** x +=b*y; **}** **}**

1.4 Comment opère le compilateur C#

C'est le compilateur C# qui fait tout le travail de recherche de la bonne méthode. Prenons un objet obj de classe Classe1, lorsque le compilateur C# trouve une instruction du genre "obj.**method1**(paramètres effectifs);", sa démarche d'analyse est semblable à celle du compilateur Delphi, il cherche dans l'ordre suivant :

- Y-a-t-il dans Classe1, une méthode qui se nomme **method1** ayant une signature identique aux paramètres effectifs ?
- si oui c'est la méthode ayant cette signature qui est appelée,
- si non le compilateur remonte dans la hierarchie des classes mères de Classe1 en posant la même question récursivement jusqu'à ce qu'il termine sur la classe Object.
- Si aucune méthode ayant cette signature n'est trouvée il signale une erreur.

Soit à partir de l'exemple précédent les instructions suivantes :
ClasseFille obj = **new** ClasseFille( );
obj.**g**(-3,8);

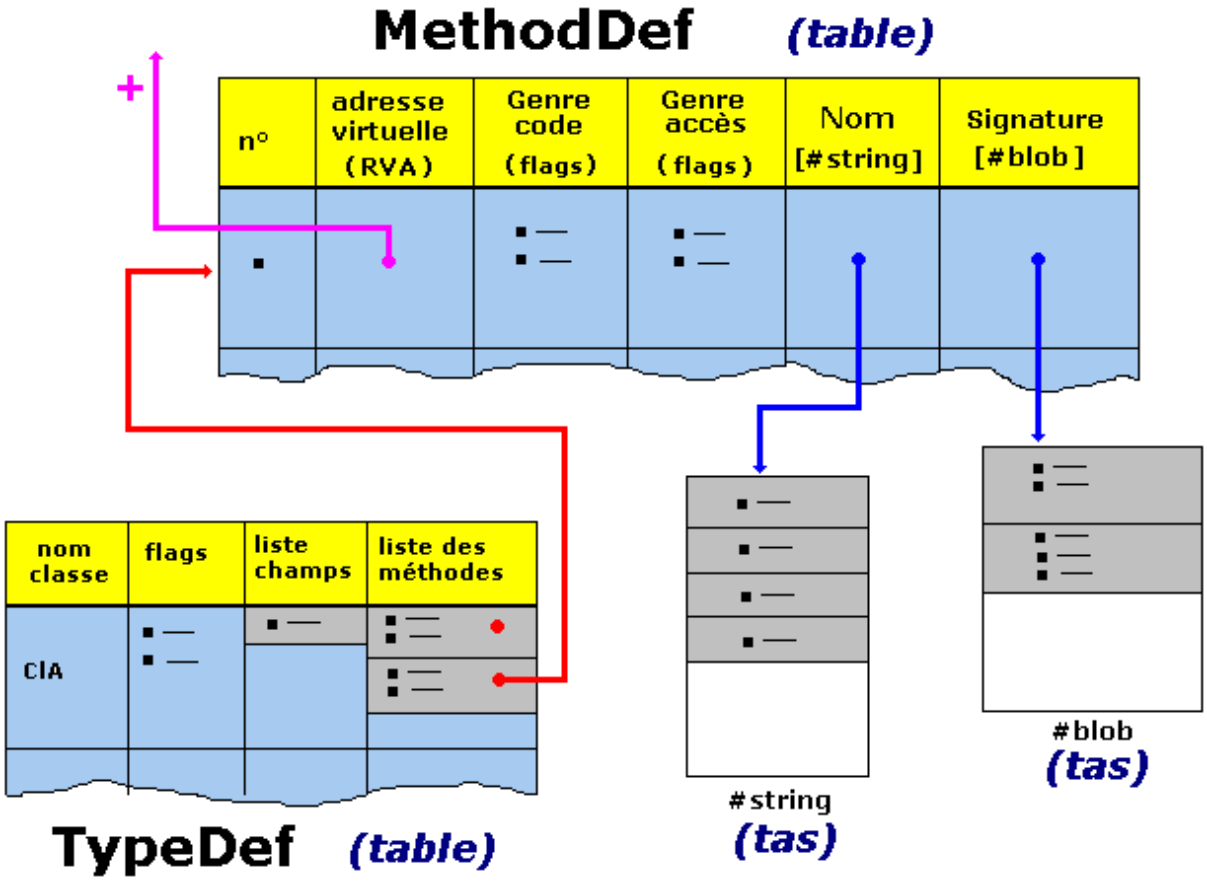
obj.**g**('h');

Le compilateur C# applique la démarche d'analyse décrite, à l'instruction "obj.**g**(-3,8);". Ne trouvant pas dans ClasseFille de méthode ayant la bonne signature (signature = deux entiers) , le compilateur remonte dans la classe mère ClasseMere et trouve une méthode " **void** g (**int** a, **int** b) " de la classe ClasseMere ayant la bonne signature (signature = deux entiers), il procède alors à l'appel de cette méthode sur les paramètres effectifs (-3,8).

Dans le cas de l'instruction obj.**g**('h'); , le compilateur trouve immédiatement dans ClasseFille la méthode " **void** g (**char** b) " ayant la bonne signature, c'est donc elle qui est appelée sur le paramètre effectif 'h'.

Le compilateur consulte les méta-données (informations de description) de l'assemblage en cours (applicationXXX.exe), plus particulièrement les métadonnées de type qui sont stockées au fur et à mesure dans de nombreuses tables.

Nous figurons ci-dessous deux tables de définition importantes relativement au polymorphisme de méthode **MethodDef** et **TypeDef** utilisées par le compilateur.



Résumé pratique sur le polymorphisme en C#

La **surcharge** (polymorphisme statique) consiste à proposer différentes signatures de la même méthode.
La **redéfinition** (polymorphisme dynamique) ne se produit que dans l'héritage d'une classe, par redéfinition (liaison dynamique) de la méthode mère avec une méthode fille (ayant ou n'ayant pas la même signature).
Le **masquage** ne se produit que dans l'héritage d'une classe, par redéfinition (liaison statique) de la méthode mère par une méthode fille (ayant la même signature).
Toute méthode est considérée à liaison statique sauf si vous la déclarez autrement.

2. Accès à la super classe en C#

2.1 Le mot clef ' base '

Nous venons de voir que le compilateur s'arrête dès qu'il trouve une méthode ayant la bonne signature dans la hiérarchie des classes, il est des cas où nous voudrions accéder à une méthode de la classe mère alors que celle-ci est redéfinie dans la classe fille. C'est un problème analogue à l'utilisation du **this** lors du masquage d'un attribut.

classe mère	classe fille

`class ClasseA`
`{`
`  public int attrA ;`
`  private int attrXA ;`
 	
`  public void meth01 ( ) {`
`  attrA = 57 ;`
`  }`
`}`
	`class ClasseB : ClasseA`
	`{`
	`  public new void meth01 ( ) {`
	`  attrA = 1000 ;`
	`  }`
	`  public void meth02 ( ) {`
	`  meth01 ( );`
	`  }`
	`}`

La méthode meth02 **( )** invoque la méthode meth01 **( )** de la classe ClasseB. Il est impossible de faire directement appel à la méthode meth01 **( )** de la classe mère ClasseA car celle-ci est masquée dans la classe fille.

```
class ClasseB : ClasseA
{
 public new void meth01 () {
   attrA = 1000 ;
 }
 public void meth02 () {
   meth01 ();
 }
}
```

Il existe en C# un mécanisme déclenché par un mot clef qui permet d'accéder à la classe mère (classe immédiatement au dessus): ce mot est **base**.

Le mot clef **base** est utilisé pour accéder à **tous les membres visibles de la classe mère** à partir d'une classe fille dérivée directement de cette classe mère (la super-classe en Java).

Ce mot clef **base** est très semblable au mot clef **inherited** de Delphi qui joue le même rôle sur les méthodes et les propriétés (il est en fait plus proche du mot clef **super** de Java car il ne remonte qu'à la classe mère), il permet l'appel d'une méthode de la classe de base qui a été substituée (masquée ou redéfinie) par une autre méthode dans la classe fille.

Exemple :

```
class ClasseA
{
 public int attrA ;
 private int attrXA ;

 public void meth01 () {
   attrA = 57 ;
 }
}
```

```
class ClasseB : ClasseA
{
 public new void meth01 () {
   attrA = 1000 ;
 }
 public void meth02 () {
   base.meth01 ();

   meth01 ();
 }
}
```

Remarques :

- Le fait d'utiliser le mot clef **base** à partir d'une méthode statique constitue une erreur.
- **base** est utile pour spécifier un constructeur de classe mère lors de la création d'instances de la classe fille.

Nous développons ci-dessous l'utilisation du mot clef base afin d'initialiser un constructeur.

2.2 Initialiseur de constructeur this et base

Semblablement à Delphi et à Java, tous les constructeurs d'instance C# autorisent l'appel d'un autre constructeur d'instance immédiatement avant le corps du constructeur, cet appel est dénommé l'initialiseur du constructeur, en Delphi cet appel doit être explicite, en C# et en Java cet appel peut être implicite.

Rappelons que comme en Java où dans toute classe ne contenant aucun constructeur, en C# **un constructeur sans paramètres par défaut** est implicitement défini :

vous écrivez votre code comme ceci :	il est complété implicitement ainsi :

`class ClasseA {`
 	
`public int attrA ;`
`public string attrStrA ;`
 	
`}`
	`class ClasseA {`
	`public int attrA ;`
	`public string attrStrA ;`
	`public ClasseA ( ) {`
	`}`
	`}`

Remarque :

Lors de l'héritage d'une classe fille, différemment à Delphi et à Java, si un constructeur d'instance C# de la classe fille **ne fait pas figurer explicitement** d'initialiseur de constructeur, c'est qu'en fait un initialiseur de constructeur ayant la forme **base( )** lui a été fourni **implicitement**.

Soit par exemple une classe ClasseA possédant 2 constructeurs :

```
class ClasseA {
 public int attrA ;
 public string attrStrA ;

 public ClasseA ( ) { /* premier constructeur */
   attrA = 57 ;
 }
 public ClasseA ( string s ) { /* second constructeur */
   attrStrA = s +"...1..." ;
 }
}
```

Soit par suite une classe fille ClasseB dérivant de ClasseA possédant elle aussi 2 constructeurs, les deux déclarations ci-dessous sont équivalentes :

Initialiseur implicite	Initialiseur explicite équivalent

`class ClasseB : ClasseA {`
 	
`/* premier constructeur */`
`public ClasseB ( ) {`
`attrStrA = "..." ;`
	`class ClasseB : ClasseA {`
	
	`/* premier constructeur */`
	`public ClasseB ( ) : base( ) {`
	`attrStrA = "..." ;`

`}`
 	
`/* second constructeur */`
`public ClasseB ( string s ) {`
`attrStrA = s ;`
`}`
`}`
	`}`
	
	`/* second constructeur */`
	`public ClasseB ( string s ) : base( ) {`
	`attrStrA = s ;`
	`}`
	`}`

Dans les deux cas le corps du constructeur de la classe fille est initialisé par un premier appel au constructeur de la classe mère ( ), en l'occurrence << **public** ClasseA **( ) ...** */* premier constructeur */* >>

Remarque :

De même pour une classe fille, C# comme Java, **tout constructeur de la classe fille appelle implicitement** et automatiquement le constructeur par défaut (celui sans paramètres) de la classe mère.

Exemple :

vous écrivez votre code comme ceci :	il est complété implicitement ainsi :

`class ClasseA {`
`public int attrA ;`
`public string attrStrA ;`
`}`
 	
`class ClasseB : ClasseA {`
 	
`}`
 	
Le constructeur de ClasseA sans paramètres est implicitement déclaré par le compilateur.
	`class ClasseA {`
	`public int attrA ;`
	`public string attrStrA ;`
	
	`public ClasseA ( ) {`
	`}`
	`}`
	
	`class ClasseB : ClasseA {`
	`public ClasseB ( ): base( ) {`
	`}`
	`}`

Si la **classe mère ne possède pas de constructeur par défaut**, le compilateur engendre un message d'erreur :

vous écrivez votre code comme ceci :	il est complété implicitement ainsi :

`class ClasseA {`
`public int attrA ;`
`public string attrStrA ;`
`public ClasseA ( int a ) {`
`}`
`class ClasseB : ClasseA {`
`public ClasseB ( ) {`
`//.....`
`}`
	`class ClasseA {`
	`public int attrA ;`
	`public string attrStrA ;`
	`public ClasseA ( int a ) {`
	`}`
	`}`
	`class ClasseB : ClasseA {`
	`public ClasseB ( ): base( ) {`
	`// ....`

} La classe de base ClasseA ne comporte qu'un seul constructeur explicite à un paramètre. Le constructeur sans paramètres n'existe que si vous le déclarez explicitement, ou bien si la classe ne possède pas de constructeur explicitement déclaré.	**}** **}** L'initialiseur implicite **base( )** renvoie le compilateur chercher dans la classe de base un constructeur sans paramètres. Or il n'existe pas dans la classe de base (ClasseA) de constructeur par défaut sans paramètres. Donc la tentative échoue !

Le message d'erreur sur la ligne " **public** ClasseB ( ) **{** ", est le suivant :
[C# Erreur] Class.cs(54): Aucune surcharge pour la méthode 'ClasseA' ne prend d'arguments '0'

Remarques :

Donc sans initialiseur explicite, **tout objet de classe fille** ClasseB est à minima et **par défaut, instancié** comme un **objet de classe de base** ClasseA.

Lorsque l'on veut **invoquer dans un constructeur** d'une classe donnée **un autre constructeur de cette même classe** étant donné que tous les constructeurs ont le même nom, il faut utiliser le mot clef **this** comme nom d'appel.

Exemple :

Reprenons la même classe ClasseA possédant 2 constructeurs et la classe ClasseB dérivant de ClasseA, nous marquons les actions des constructeurs par une chaîne indiquant le numéro du constructeur invoqué ainsi que sa classe :

```
class ClasseA {
  public int attrA ;
  public string attrStrA ;

  public ClasseA ( ) { /* premier constructeur */
    attrA = 57 ;
  }
  public ClasseA ( string s ) { /* second constructeur */
    attrStrA = s +"...classeA1..." ;
  }
}
```

Ci-dessous la ClasseB écrite de deux façons équivalentes :

avec initialiseurs implicites-explicites	avec initialiseurs explicites équivalents
class ClasseB : ClasseA **{** */* premier constructeur */* **public** ClasseB **( ) {**	**class** ClasseB : ClasseA **{** */* premier constructeur */* **public** ClasseB **( ) : base( ) {**

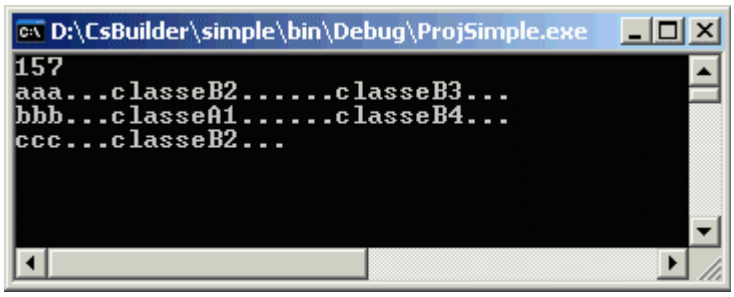
```
    attrA = 100+attrA  ;
  }

 /* second constructeur */
 public  ClasseB ( string  s )  {
    attrStrA = attrStrA +s+"...classeB2..." ;
  }

/* troisième constructeur */
 public  ClasseB ( int x , string  ch ) : this( ch )  {
    attrStrA = attrStrA+"...classeB3..."  ;
  }

/* quatrième constructeur */
 public  ClasseB ( char x , string  ch ) : base( ch )  {
    attrStrA = attrStrA+"...classeB4..."  ;
  }
}
```

```
    attrA = 100+attrA  ;
  }

 /* second constructeur */
 public  ClasseB ( string  s ): base( )  {
    attrStrA = attrStrA +s+"...classeB2..." ;
  }

/* troisième constructeur */
 public  ClasseA ( int x , string  ch ) : this( ch )  {
    attrStrA = attrStrA+"...classeB3..."  ;
  }

/* quatrième constructeur */
 public  ClasseB ( char x , string  ch ) : base( ch )
{
    attrStrA = attrStrA+"...classeB4..."  ;
  }
}
```

Créons quatre objets de ClasseB, chacun avec l'un des 4 constructeurs de la ClasseB :

```
class MaClass    {
      static void Main(string[] args)   {
        int x=68;
        ClasseB ObjetB= new ClasseB( );
        System.Console.WriteLine(ObjetB.attrA);
        ObjetB= new ClasseB(x,"aaa");
        System.Console.WriteLine(ObjetB.attrStrA);
        ObjetB= new ClasseB((char)x,"bbb");
        System.Console.WriteLine(ObjetB.attrStrA);
        ObjetB= new ClasseB("ccc");
        System.Console.WriteLine(ObjetB.attrStrA);
        System.Console.ReadLine();
      }
   }
```

Voici le résultat console de l'exécution de ce programme :



Explications :

`public ClasseB ( ) {`
`  attrA = 100+attrA ;`
`}`
 	
`ClasseB ObjetB= new ClasseB( );`
`System.Console.WriteLine(ObjetB.attrA);`
	C# sélectionne la signature du premier constructeur de la ClasseB (le constructeur sans paramètres).
	
	C# appelle d'abord implicitement le constructeur sans paramètre de la classe mère (: base( ))

	public ClasseA **( ) {** attrA = 57 **;** **}** Le champ attrA vaut 57, puis C# exécute le corps du constructeur : attrA = 100+attrA **;** attrA vaut 100+57 = 157
public ClasseB **(** string s **) {** attrStrA = attrStrA +s+**"...classeB2..." ;** **}** **public** ClasseB **(** int x , string ch **) : this(ch) {** attrStrA = attrStrA+**"...classeB3..." ;** **}** ObjetB= **new** ClasseB(x,"aaa"); System.Console.WriteLine(ObjetB.attrStrA);	C# sélectionne la signature du troisième constructeur de la ClasseB (le constructeur avec paramètres : int x , string ch). C# appelle d'abord explicitement le constructeur local de la classeB avec un paramètre de type string (le second constructeur de la ClasseB) s = "aaa" ; **public** ClasseB **(** string s **) {** attrStrA = attrStrA +s+**"...classeB2..." ;** **}** Le champ attrStrA vaut **"aaa...classeB2..."**, puis C# exécute le corps du constructeur : attrStrA = attrStrA+**"...classeB3..." ;** attrStrA vaut **"aaa...classeB2.....classeB3..."**
public ClasseB **(** char x , string ch **) : base(ch)** **{** attrStrA = attrStrA+**"...classeB4..." ;** **}** ObjetB= **new** ClasseB((**char**)x,"bbb"); System.Console.WriteLine(ObjetB.attrStrA);	C# sélectionne la signature du quatrième constructeur de la ClasseB (le constructeur avec paramètres : char x , string ch). C# appelle d'abord explicitement le constructeur de la classe mère (de base) classeA avec un paramètre de type string (ici le second constructeur de la ClasseA) s = "bbb" ; **public** ClasseA **(** string s **) {** attrStrA = s +**"...classeA1..." ;** **}** Le champ attrStrA vaut **"bbb...classeA1..."** puis C# exécute le corps du constructeur : attrStrA = attrStrA+**"...classeB4..." ;** attrStrA vaut **"bbb...classeA1.....classeB4..."**

La dernière instanciation : ObjetB= **new** ClasseB("ccc"); est strictement identique à la première mais avec appel au second constructeur.

2.3 Comparaison de construction C#, Delphi et Java

Exemple classe mère :

C#	Java

```
class ClasseA {
 public int attrA ;
 public string attrStrA ;

 public ClasseA ( ) {
   attrA = 57 ;
 }
 public ClasseA ( string s ) {
   attrStrA = s +"...classeA1..." ;
 }
}
```

```
class ClasseA {
 public int attrA ;
 public String attrStrA = "" ;

 public ClasseA ( ) {
   attrA = 57 ;
 }
 public ClasseA ( String s ) {
   attrStrA = s +"...classeA1..." ;
 }
}
```

C#

```
class ClasseA {
 public int attrA ;
 public string attrStrA ;

 public ClasseA ( ) {
   attrA = 57 ;
 }
 public ClasseA ( string s ) {
   attrStrA = s +"...classeA1..." ;
 }
}
```

Delphi

```
ClasseA = class
 public
   attrA : integer ;
  attrStrA: string ;
  constructor Creer;overload;
  constructor Creer(s:string); overload;
 end;

constructor ClasseA.Creer
 begin
   attrA := 57 ;
 end;
constructor ClasseA.Creer(s:string);
 begin
   attrStrA := s +'...classeA1...' ;
 end;
```

Exemple classe fille :

C#

```
class ClasseB : ClasseA {

  /* premier constructeur */
   public ClasseB ( ) {
     attrA = 100+attrA ;
   }

  /* second constructeur */
  public ClasseB ( string s ) {
     attrStrA = attrStrA +s+"...classeB2..." ;
   }


 /* troisième constructeur */
  public ClasseB ( int x , string ch ) : this(
 ch ) {
     attrStrA = attrStrA+"...classeB3..." ;
   }
```

Java

```
class ClasseB extends ClasseA {

 /* premier constructeur */
 public ClasseB ( ) {
  super( ) ;
  attrA = 100+attrA ;
 }

   /* second constructeur */
   public ClasseB ( String s ) {
     super( ) ;
     attrStrA = attrStrA +s+"...classeB2..." ;
   }

  /* troisième constructeur */
   public ClasseB ( int x , String ch ) {
     this( ch ) ;
     attrStrA = attrStrA+"...classeB3..." ;
```

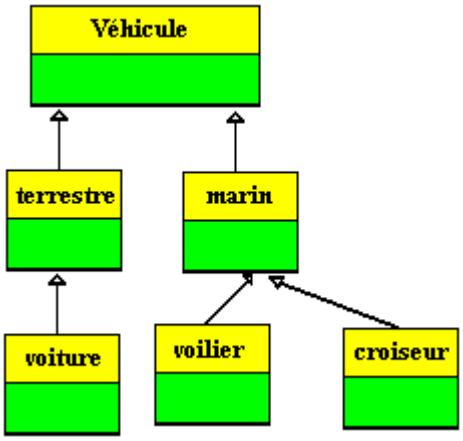
/* quatrième constructeur */
public ClasseB (char x , string ch) :
base(ch) {
attrStrA = attrStrA+"...classeB4..." ;
}
 	
}
	}
	
	/* quatrième constructeur */
	public ClasseB (char x , String ch) {
	super(ch) ;
	attrStrA = attrStrA+"...classeB4..." ;
	}
	}

C#	Delphi

class ClasseB : ClasseA
{
 	
/* premier constructeur */
public ClasseB ( )
{
attrA = 100+attrA ;
}
 	
/* second constructeur */
public ClasseB (string s)
{
attrStrA = attrStrA +s+"...classeB2..." ;
}
 	
/* troisième constructeur */
public ClasseB (int x , string ch) : this(ch)
{
attrStrA = attrStrA+"...classeB3..." ;
}
 	
/* quatrième constructeur */
public ClasseB (char x , string ch) : base(ch)
{
attrStrA = attrStrA+"...classeB4..." ;
}
 	
}
	ClasseB = class(ClasseA)
	public
	constructor Creer;overload;
	constructor Creer(s:string); overload;
	constructor Creer(x:integer;ch:string); overload;
	constructor Creer(x:char;ch:string); overload;
	end;
	/* premier constructeur */
	constructor ClasseB.Creer;
	begin
	inherited ;
	attrA := 100+attrA ;
	end;
	
	/* second constructeur */
	constructor ClasseB.Creer(s:string);
	begin
	inherited Creer ;
	attrStrA := attrStrA +s+'...classeB2...' ;
	end;
	
	/* troisième constructeur */
	constructor ClasseB.Creer(x:integer;ch:string);
	begin
	Creer(ch) ;
	attrStrA := attrStrA+'...classeB3...' ;
	end;
	
	/* quatrième constructeur */
	constructor ClasseB.Creer(x:integer;ch:string);
	begin
	inherited Creer(ch) ;
	attrStrA := attrStrA+'...classeB4...' ;
	end;

2.4 Traitement d'un exercice complet

soit une hiérarchie de classe de véhicules :

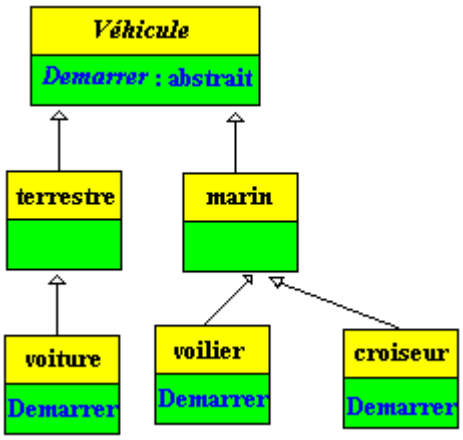


syntaxe de base :

class Vehicule **{** **}** **class** Terrestre :Vehicule **{** **}** **class** Marin :Vehicule **{** **}**	**class** Voiture : Terrestre **{** **}** **class** Voilier : Marin **{** **}** **class** Croiseur : Marin **{** **}**

Supposons que la classe Véhicule contienne 3 méthodes, qu'elle n'implémente pas la méthode **Demarrer** qui est alors abstraite, qu'elle fournit et implante à vide la méthode "**RépartirPassagers**" de répartition des passagers à bord du véhicule, qu'elle fournit aussi et implante à vide une méthode "**PériodicitéMaintenance**" renvoyant la périodicité de la maintenance obligatoire du véhicule.

La classe Véhicule est abstraite : car la méthode **Demarrer** est abstraite et sert de "modèle" aux futures classes dérivant de Véhicule. Supposons que l'on implémente le comportement précis du genre de démarrage dans les classes **Voiture** , **Voilier** et **Croiseur** .



Dans cette hiérarchie, les classes **Terrestre** et **Marin** héritent de la classe **Vehicule,** mais

n'implémentent pas la méthode abstraite **Démarrer,** ce sont donc par construction des classes abstraites elles aussi. Elles implantent chacune la méthode "**RépartirPassagers**" (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...) et la méthode "**PériodicitéMaintenance**" (fonction du nombre de km ou miles parcourus, du nombre d'heures d'activités,...)

Les classes **Voiture** , **Voilier** et **Croiseur** savent par héritage direct comment répartir leur éventuels passagers et quand effectuer une maintenance, chacune d'elle implémente son propre comportement de démarrage.

Quelques implantations en C#

Une implémentation de la classe Voiture avec des méthodes non virtuelles *(Version-1)* :

abstract class Vehicule **{** **public abstract void** Demarrer( ); **public void** RépartirPassagers( ){} **public void** PériodicitéMaintenance( ){} **}**	La méthode **Demarrer** de la classe Vehicule est abstraite et donc automatiquement virtuelle (à liaison dynamique). Les méthodes RépartirPassagers et PériodicitéMaintenance sont concrètes mais avec un corps vide. Ces deux méthodes sont non virtuelles (à liaison statique)

abstract class Terrestre : Vehicule **{**
public new void RépartirPassagers( ){
//...}
public new void PériodicitéMaintenance( ){
//...}
}
	La classe Terrestre est abstraite car elle n'implémente pas la méthode abstraite **Demarrer**.
	
	Les deux méthodes déclarées dans la classe Terrestre masquent chacune la méthode du même nom de la classe Vehicule (d'où l'utilisation du mot clef **new**)

class Voiture : Terrestre **{** **public override void** Demarrer( ){ //...} **}**	La classe Voiture est la seule à être instanciable car toutes ses méthodes sont concrètes : Elle hérite des 2 méthodes implémentées de la classe Terrestre et elle implante (redéfinition avec **override**) la méthode abstraite de l'ancêtre.

La même implémentation de la classe Voiture avec des méthodes virtuelles *(Version-2)*:

abstract class Vehicule **{** **public abstract void** Demarrer( ); **public virtual void** RépartirPassagers( ){} **public virtual void** PériodicitéMaintenance( ){} **}**	La méthode **Demarrer** de la classe Vehicule est abstraite et donc automatiquement virtuelle (à liaison dynamique). Les méthodesRépartirPassagers et PériodicitéMaintenance sont concrètes mais avec un corps vide. Ces deux méthodes sont maintenant virtuelles (à liaison dynamique)

abstract class Terrestre : Vehicule **{**
public override void RépartirPassagers( ){
//...}
public override void PériodicitéMaintenance( ){
//...}
	La classe Terrestre est abstraite car elle n'implémente pas la méthode abstraite **Demarrer**.
	
	Les deux méthodes déclarées dans la classe Terrestre redéfinissent chacune la méthode du même nom de la

}	classe Vehicule (d'où l'utilisation du mot clef **override**)
class Voiture : Terrestre **{** **public override void** Demarrer( ){ //...} **}**	La classe Voiture est la seule à être instanciable car toutes ses méthodes sont concrètes : Elle hérite des 2 méthodes implémentées de la classe Terrestre et elle implante (redéfinition avec **override**) la méthode abstraite de l'ancêtre.

Supposons que les méthodes non virtuelles RépartirPassagers et PériodicitéMaintenance sont implantées complètement dans la classe Vehicule, puis reprenons la classe Terrestre en masquant ces deux méthodes :

```
abstract class Vehicule  {
      public abstract void Demarrer( );
      public void RépartirPassagers( ){
      //....}
      public void PériodicitéMaintenance( ){
      //....}
}

abstract class Terrestre : Vehicule  {
      public new void RépartirPassagers( ){
     //...}
      public new void PériodicitéMaintenance( ){
      //...}
}
```

Question

Nous voulons qu'un véhicule Terrestre répartisse ses passagers ainsi :
1°) d'abord comme tous les objets de classe Vehicule,
2°) ensuite qu'il rajoute un comportement qui lui est propre

Réponse

La méthode RépartirPassagers est non virtuelle, elle masque la méthode mère du même nom, si nous voulons accèder au comportement de base d'un véhicule, il nous faut utiliser le mot clef **base** permettant d'accèder aux membres de la classe mère :

```
abstract class Terrestre : Vehicule  {
      public new void RépartirPassagers( ){
         base.RépartirPassagers( );  //... 1°; comportement du parent
         //... 2° comportement propre
      }
      public new void PériodicitéMaintenance( ){
      //...}
}
```

Il est conseillé au lecteur de reprendre le même schéma et d'implanter à l'identique les autres classe de la hiérarchie pour la branche des véhicules **Marin**.

2.5 Destructeur – Finaliseur

Les classes de C# peuvent posséder un finaliseur (syntaxe semblable au destructeur de C++) mais comme la gestion de la mémoire est automatiquement assurée par le CLR, l'usage d'un finaliseur n'est pas le même que celui qui en est fait avec Delphi ou C++ dans lesquels la récupération de la mémoire est à la charge du développeur.

Les structs ne peuvent pas possèder de finaliseur (ou destructeur)

Propriétés et usage des finaliseurs avec C# :

Une classe C# ne peut posséder qu'un seul finaliseur.

Un finaliseur ne peut pas être redéfini ni surchargé.

Un finaliseur n'a pas de paramètre.

Un destructeur est appelé **automatiquement** par le CLR lors de la destruction effective de l'objet par le CLR, il ne peut pas être appelé directement.

De ces propriétés il découle qu'un destructeur peut être utilisée par le développeur pour libérer certaines ressources non gérées par le CLR (fermeture de fichier, fermeture de connexion,...) lorsque le programme ne les utilise plus.

Exemple :

```
class classeB : classeA
{
    // constructeur :
    public classeB( )
    {
        .....
    }
    // destructeur :
    ~classeB( )
    {
        Console.WriteLine("finalizeur de la classeB ");
    }
}
```

A chaque fois qu'un objet de classeA est détruit par le CLR, le message "finalizeur de la classeB" apparaîtra sur la console. Il est entendu qu'il ne faut pas se servir de la programmation des destructeurs (finaliseurs) pour gérer un programme d'une manière synchrone puisque le programme n'a aucun contrôle sur la façon dont les objets sont libérés par le CLR qui utilise son propre algorithme de garbage collection et de restitution de mémoire.

Enfin, un destructeur appelle automatiquement le destructeur de la classe mère et ainsi de suite récursivement jusqu'à la classe object. Dans l'exemple précédent, le destructeur de la classeB, après écriture du message "finalizeur de la classeB" , appelle le destructeur de la classeA.

Polymorphisme et interfaces en

Plan général:

Rappels sur la notion d'interface

1.Concepts et vocabulaire d'interface en C#

- les interfaces peuvent constituer des hiérarchies et hériter entre elles
- la construction d'un objet necessite une classe implémentant l'interface
- les implémentations d'un membre d'interface sont en général public
- les implémentations explicites d'un membre d'interface sont spéciales

1.1 Spécification d'un exemple complet
1.1.A Une classe abstraite
1.1.B Une interface
1.1.C Une simulation d'héritage multiple
1.1.D Encore une classe abstraite, mais plus concrète
1.1.E Une classe concrète

1.2 Implantation en C# de l'exemple
1.2.A La classe abstraite
1.2.B L'interface
1.2.C La simulation d'héritage multiple
1.2.D La nouvelle classe abstraite
1.2.E La classe concrète

2. Analyse du code de liaison de la solution précédente
2.1 Le code de la classe Vehicule
2.2 Le code de l'interface IVehicule
2.3 Le code de la classe UnVehicule
2.4 Le code de la classe Terrestre
2.5 Le code de la classe Voiture

3. Cohérence de C# entre les notions de classe et d'interface

- une classe peut implémenter plusieures interfaces
- les interfaces et les classes respectent les mêmes règles de polymorphisme
- les conflits de noms dans les interfaces

Rappels essentiels sur la notion d'interface

- Les interfaces ressemblent aux classes abstraites : elles contiennent des membres **spécifiant certains comportements sans les implémenter**.

- Les classes abstraites et les interfaces se différencient principalement par le fait qu'**une classe peut implémenter un nombre quelconque d'interfaces**, alors qu'une classe abstraite ne peut hériter que d'**une seule classe** abstraite ou non.

- Une **interface** peut servir à représenter des comportements d'héritage multiple.

Quelques conseils généraux prodigués par des développeurs professionnels (microsoft, Borland, Sun) :

- *Les interfaces bien conçues sont plutôt petites et indépendantes les unes des autres.*
- *Un trop grand nombre de fonctions rend l'interface peu maniable.*
- *Si une modification s'avère nécessaire, une nouvelle interface doit être créée.*
- *Si la fonctionnalité que vous créez peut être utile à de nombreux objets différents, faites appel à une interface.*
- *Si vous créez des fonctionnalités sous la forme de petits morceaux concis, faites appel aux interfaces.*
- *L'utilisation d'interfaces permet d'envisager une conception qui sépare la manière d'utiliser une classe de la manière dont elle est implémentée.*
- *Deux classes peuvent partager la même interface sans descendre nécessairement de la même classe de base.*

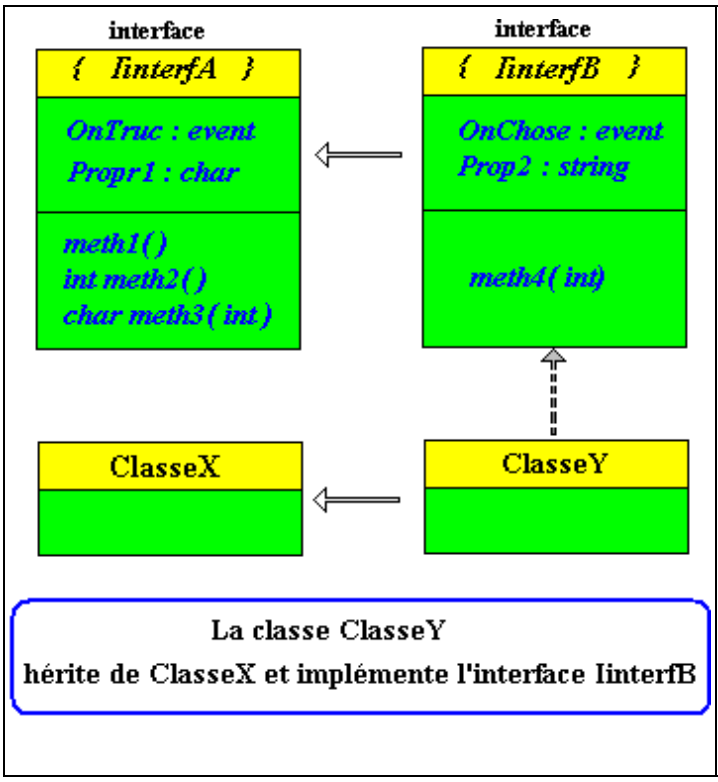
1. Vocabulaire et concepts en C#

- Une interface C# est un contrat, elle peut contenir des **propriétés**, des **méthodes** , des **événements** ou des **indexeurs**, mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une interface **ne** peut **pas** contenir des méthodes déjà implémentées.
- Une interface ne contient que des signatures (**propriétés**, **méthodes**).
- **Tous les membres** d'une interface sont automatiquement **public**.
- Une interface est héritable.
- On peut construire une hiérarchie d'interfaces.
- Pour pouvoir construire un objet à partir d'une interface, il faut définir une classe non abstraite implémentant **tous** les membres de l'interface.

Les interfaces peuvent constituer des hiérarchies et hériter entre elles

soient l'interface **IinterfA** et l'interface **IinterfB** héritant de **IinterfA** . On pourra employer aussi le vocable d'étendre au sens où l'interface dérivée **IinterfB** "étend" le contrat (augmente le nombre de membres contractuels) de l'interface **IinterfA** .

Dans tous les cas il faut une classe pour implémenter ces contrats :



code C# :

```
interface  IinterfA {
  event  TypEvent1 OnTruc ;  // événement
   char  Prop1    // propriété
   {      get ;      set ;   }
  void  meth1 ( ); //  méthode
  int  meth2 ( ); //  méthode
  char  meth3 ( int x ); //  méthode
}

interface  IinterfB :  IinterfA {
  event  TypEvent2 OnChose ;  //
événement
   string  Prop2    //  propriété
   {      get ;     }
  void  meth4 ( ); //  méthode
}
```

La construction d'un objet nécessite une classe implémentant l'interface

La classe ClasseY doit implémenter tous les 8 membres provenant de l'héritage des interfaces : les 2 événements **OnTruc** et **Onchose**, les 2 propriétés **Prop1** et **Prop2**, et enfin les 4 méthodes **meth1**, ... , **meth4** . La classe ClasseY est une classe concrète (instanciable), un objet de cette classe possède en particulier tous les membres de l'interface IinterfB (et donc IinterfA car IinterfB hérite de IinterfA)

```
class ClasseY : ClasseX , IinterfB {
  //  ... implémente tous les membres de InterfB
}
//  construction et utilisation d'un objet :
ClasseY Obj = new ClasseY( ) ;
Obj.Prop1 =  'a' ; //  propriété  héritée de  InterfA
string s = Obj.prop2 ; //  propriété  héritée de  InterfB
Obj.meth1( ) ; //  méthode  héritée de  InterfA
etc ...
```

Si, ClasseY n'implémente par exemple que 7 membres sur les 8 alors C# considère que c'est une classe abstraite et vous devez la déclarer abstract :

```
abstract class ClasseY : ClasseX , IinterfB {
  //  ...n' implémente que certains membres de InterfB
}
class ClasseZ : ClasseY  {
  //  ... implémente le reste des membres de InterfB
}
//  ... construction d'un objet :
ClasseZ Obj = new ClasseZ( ) ;
```

```
Obj.Prop1 =  'a' ; // propriété héritée de InterfA
string s = Obj.prop2 ; // propriété héritée de InterfB
Obj.meth1( ) ; // méthode héritée de InterfA
etc ...
```

Les implémentations des membres d'une interface sont en général public

Par défaut sans déclaration explicite, les membres (indexeurs, propriétés, événements, méthodes) d'une interface ne nécessitent pas de qualificateurs de visibilité car ils sont automatiquement déclarés par C# comme étant de visibilité public, contrairement à une classe ou par défaut les membres sont du niveau assembly.

Ce qui signifie que toute classe qui implémente un membre de l'interface doit obligatoirement le qualifier de **public** sous peine d'avoir un message d'erreur du compilateur, dans cette éventualité le membre devient un membre d'instance. Comme la signature de la méthode n'est qu'un contrat, le mode de liaison du membre n'est pas fixé; la classe qui implémente le membre peut alors choisir de l'implémenter soit **en liaison statique**, soit **en liaison dynamique**.

Soient une interface IinterfA et une classe ClasseX héritant directement de la classe Object et implémentant cette interface, ci-dessous les deux seules implémentations possibles d'une méthode avec un rappel sur les redéfinitions possibles dans des classes descendantes :

```
interface  IinterfA {
  void  meth1 ( ); // méthode de l'interface
}
```

Implémentation en liaison précoce	Implémentation en liaison tardive
meth1 devient une méthode d'instance	
class ClasseX : IinterfA **{** **public void** meth1 **( ){ ... }** **}**	**class** ClasseX : IinterfA **{** **public virtual void** meth1 **( ){ ... }** **}**
Redéfinitions possibles	**Redéfinitions possibles**
class ClasseY : ClasseX **{** **public new void** meth1 **( ){ ... }** *masque statiquement celle de ClasseX* **}** **class** ClasseZ : ClasseX **{** **public new virtual void** meth1 **( ){ ... }** *masque dynamiquement celle de ClasseX* **}**	**class** ClasseY : ClasseX **{** **public new void** meth1 **( ){ ... }** *masque statiquement celle de ClasseX* **}** **class** ClasseZ : ClasseX **{** **public new virtual void** meth1 **( ){ ... }** *masque dynamiquement celle de ClasseX* **}** **class** ClasseT : ClasseX **{** **public override void** meth1 **( ){ ... }** *redéfinit dynamiquement celle de ClasseX* **}**

Les implémentations explicites des membres d'une interface sont spéciales

Une classe qui implémente une interface peut aussi implémenter de façon **explicite** un membre de cette interface. Lorsqu'un membre est implémenté de façon explicite (le nom du membre

est préfixé par le nom de l'interface : **InterfaceXxx.NomDuMembre**), il n'est pas accessible via une référence de classe, il est alors **invisible** à tout objet instancié à partir de la classe où il est défini. Un membre implémenté de façon explicite n'est donc pas un membre d'instance.

Pour utiliser un membre d'interface implémenté de manière explicite, il faut utiliser une référence sur cette interface et non une référence de classe; il devient visible uniquement à travers une référence sur l'interface.

Nous reprenons le même tableau de différentes implémentations de la méthode **void** meth1 **( )** en ajoutant une nouvelle méthode **void** meth2 **(** int x **)** que nous **implémentons explicitement** dans les classes dérivées :

```
interface  IinterfA {
   void  meth2 ( int  x ); //  méthode de l'interface
   void  meth1 ( ); //  méthode de l'interface
}
```

Nous implémentons l'interface IinterfA dans la classe ClasseX :

1°) nous implémentons **explicitement void** meth2 **(** int x **)**,
2°) nous implémentons **void** meth1 **( )** en méthode **virtuelle**.

```
interface  IinterfA {
   void  meth2 ( int  x ); //  méthode de l'interface
   void  meth1 ( ); //  méthode de l'interface
}
class ClasseX :  IinterfA {
  void  IinterfA.meth2 ( int  x ){ ... }
  public virtual void  meth1 ( ){ ... }
}
```

Comprenons bien que la classe ClasseX ne possède pas à cet instant une méthode d'instance qui se nommerait meth2, par exemple si dans la méthode virtuelle meth1 nous utilisons le paramètre implicite **this** qui est une référence à la future instance, l'audit de code de C#Builder nous renvoie 7 méthodes comme visibles (6 provenant de la classe mère Object et une seule provenant de ClasseX), la méthode IinterfA.meth2 n'est pas visible :

```
interface  IinterfA {
    void  meth2 ( int  x ); //  méthode de l'interface
    void  meth1 ( ); //  méthode de l'interface
}
class ClasseX :  IinterfA {
  void  IinterfA.meth2 ( int  x ){  }
  public virtual void  meth1 ( ){
  this.}
}
```

Method	**meth1**
Destructor	**Finalize**
Method	**GetType**
Method	**Equals**
Method	**MemberwiseClone**
Method	**ToString**
Method	**GetHashCode**

Lorsque l'on instancie effectivement un objet de classe ClasseX, cet objet ne voit comme méthode provenant de ClasseX que la méthode meth1 :

```
class Test{
 void methTest(){
      ClasseX Obj = new ClasseX();
      Obj.
 }
}
```

Method	**meth1**
Method	**GetType**
Method	**Equals**
Method	**ToString**
Method	**GetHashCode**

La méthode meth2 implémentée explicitement en IinterfA.meth2 devient visible uniquement si l'on utilise une référence sur l'interface IinterfA, l'exemple ci-dessous montre qu'alors les deux méthodes meth1 et meth2 sont visibles :

```
interface  IinterfA {
    void  meth2 ( int  x ); //  méthode de l'interface
    void  meth1 ( ); //  méthode de l'interface
}
class ClasseX :  IinterfA {
  void  IinterfA.meth2 ( int  x ){  }
  public virtual void  meth1 ( ){ }
}
class Test{
 void methTest(){
      IinterfA Obj = new ClasseX();
      Obj.
 }
}
```

Method	**meth2**
Method	**meth1**

L'audit de code de Visual C# fournit plus de précision directement :

```
interface  IinterfA {
  void  meth2 ( int  x ); //  méthode de l'interface
  void  meth1 ( ); //  méthode de l'interface
}
class ClasseX :  IinterfA   {
  void  IinterfA.meth2 ( int  x ){  }
  public virtual void  meth1 ( ){ }
}
class Test  {
  void methTest()   {
    IinterfA Obj = new ClasseX();
    Obj.
    }
}
```

Equals
GetHashCode
GetType
meth1
meth2
ToString

void IinterfA.meth2 (int x)

Nous voyons bien que la méthode est qualifiée avec sa signature dans IinterfA, voyons dans l'exemple ci-dessous que nous pouvons déclarer une méthode d'instance ayant la même signature que la méthode explicite, voir même de surcharger cette méthode d'instance sans que le compilateur C# n'y voit de conflit car la méthode explicite n'est pas rangé dans la table des méthodes d'instances de la classe :

```
class ClasseX :  IinterfA {
 void  IinterfA.meth2 ( int  x ){ ... }   //méthode de IinterfA implémentée explicitement
 public virtual void  meth2 ( int  x ){ ... } //méthode de la ClasseX surchargée
 public virtual void  meth2 ( char  c ){ ... } //méthode de la ClasseX surchargée
 public virtual void  meth1 ( ){ ... } //méthode de IinterfA implémentée virtuellement
}
```

```
interface  IinterfA {
  void  meth2 ( int  x ); //  méthode de l'interface
  void  meth1 ( ); //  méthode de l'interface
}
class ClasseX :  IinterfA   {
  void  IinterfA.meth2 ( int  x ){  }
  public virtual void  meth1 ( ){ }
  public virtual void  meth2 (char c){ }
  public virtual void  meth2 (int x){ }
}
class Test  {
  void methTest()   {
    ClasseX Obj1 = new ClasseX();
    IinterfA Obj2 = new ClasseX();
    Obj1.
    }
}
```

Equals
GetHashCode
GetType
meth1
meth2
ToString

void ClasseX.meth2 (char c) (+ 1 surcharges)

La référence Obj1 peut appeller les deux surcharges de la méthode d'instance meth2 de la classe ClasseX :

```
class Test  {
  void methTest()   {
    ClasseX Obj1 = new ClasseX();
    IinterfA Obj2 = new ClasseX();
    Obj1.meth2 (
    1 sur 2  void ClasseX.meth2 (char c)
}
```

```
class Test  {
  void methTest()   {
    ClasseX Obj1 = new ClasseX();
    IinterfA Obj2 = new ClasseX();
    Obj1.meth2 (
    2 sur 2  void ClasseX.meth2 (int x)
}
```

La référence Obj2 sur IinterfA fonctionne comme nous l'avons montré plus haut, elle ne peut voir de la méthode meth2 que son implémentation explicite :

```
class Test  {
  void methTest()   {
    ClasseX Obj1 = new ClasseX();
    IinterfA Obj2 = new ClasseX();
    Obj2.
    }
}
```

Equals
GetHashCode
GetType
meth1
meth2
ToString

void IinterfA.meth2 (int x)

Cette fonctionnalité d'implémentation explicite spécifique à C# peut être utilisée dans au moins deux cas utiles au développeur :

- Lorsque vous voulez qu'un membre (une méthode par exemple) implémenté d'une interface soit privé dans une classe pour toutes les instances de classes qui en dériveront, l'implémentation explicite vous permet de rendre ce membre (cette méthode) inaccessible à tout objet.
- Lors d'un conflit de noms si deux interfaces possèdent un membre ayant la même signature et que votre classe implémente les deux interfaces.

1.1 Spécification d'un exemple complet

Utilisons la notion d'interface pour fournir un polymorphisme à une hiérarchie de classe de véhicules fondée sur une **interface**. :

Soit au départ une classe abstraite **Vehicule** et une interface **IVehicule.**

1.1.A) Une classe abstraite

La classe abstraite **Vehicule** contient trois méthodes :

classe abstraite *Vehicule* + *Demarrer( )* + RépartirPassagers ( ) + PériodicitéMaintenance( )	• La méthode **Démarrer** qui est abstraite. • La méthode **RépartirPassagers** de répartition des passagers à bord du véhicule, implantée avec un corps vide. • La méthode **PériodicitéMaintenance** renvoyant la périodicité de la maintenance obligatoire du véhicule, implantée avec un corps vide.

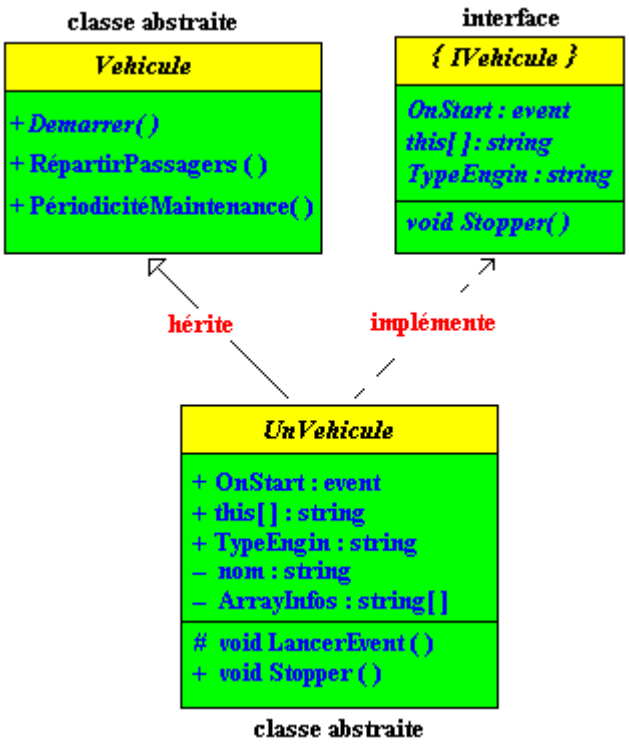
1.1.B) Une interface

Afin d'utiliser les possibilités de C#, l'interface **IVehicule** propose un contrat d'implémentation pour un événement, un indexeur, une propriété et une méthode :

interface *{ IVehicule }* *OnStart : event* *this[ ] : string* *TypeEngin : string* *void Stopper( )*	• L'événement **OnStart** est de type délégué (on construit un type délégué Starting( ) spécifique pour lui) et se déclenchera au démarrage du futur véhicule, • L'indexeur **this** [int] est de type string et permettra d'accéder à une liste indicée d'informations sur le futur véhicule, • La propriété **TypeEngin** est en lecture et écriture et concerne le type du futur véhicule dans la marque, • La méthode **Stopper( )** indique comment le futur véhicule s'immobilisera.

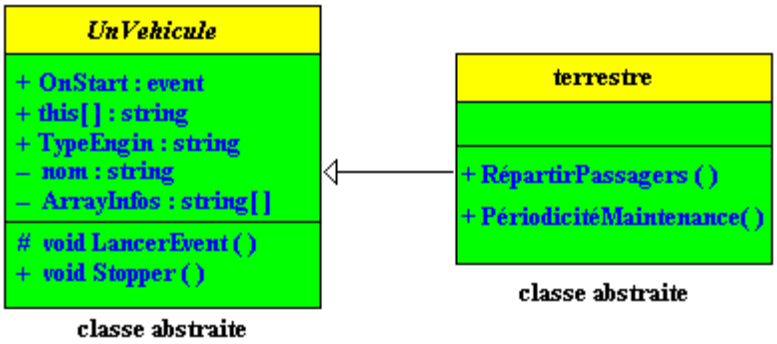
1.1.C) Une simulation d'héritage multiple

Nous souhaitons construire une classe abstraite **UnVehicule** qui "hérite" à la fois des fonctionnalités de la classe **Vehicule** et de celles de l'interface **IVehicule**. Il nous suffit en C# de faire hériter la classe **UnVehicule** de la classe **Vehicule** , puis que la classe **UnVehicule** implémente les propositions de contrat de l'interface **IVehicule** :



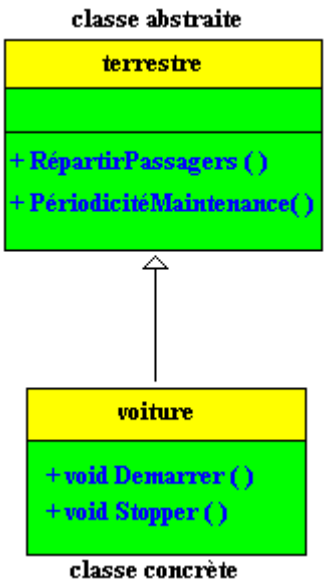
1.1.D) Encore une classe abstraite, mais plus "concrète"

Nous voulons maintenant proposer une spécialisation du véhicule en créant une classe abstraite **Terrestre** , base des futurs véhicules terrestres. Cette classe implantera de façon explicite la méthode **RépartirPassagers** de répartition des passagers et la méthode **PériodicitéMaintenance** renvoyant la périodicité de la maintenance. Cette classe **Terrestre** reste abstraite car elle ne fournit pas l'implémentation de la méthode **Demarrer** :

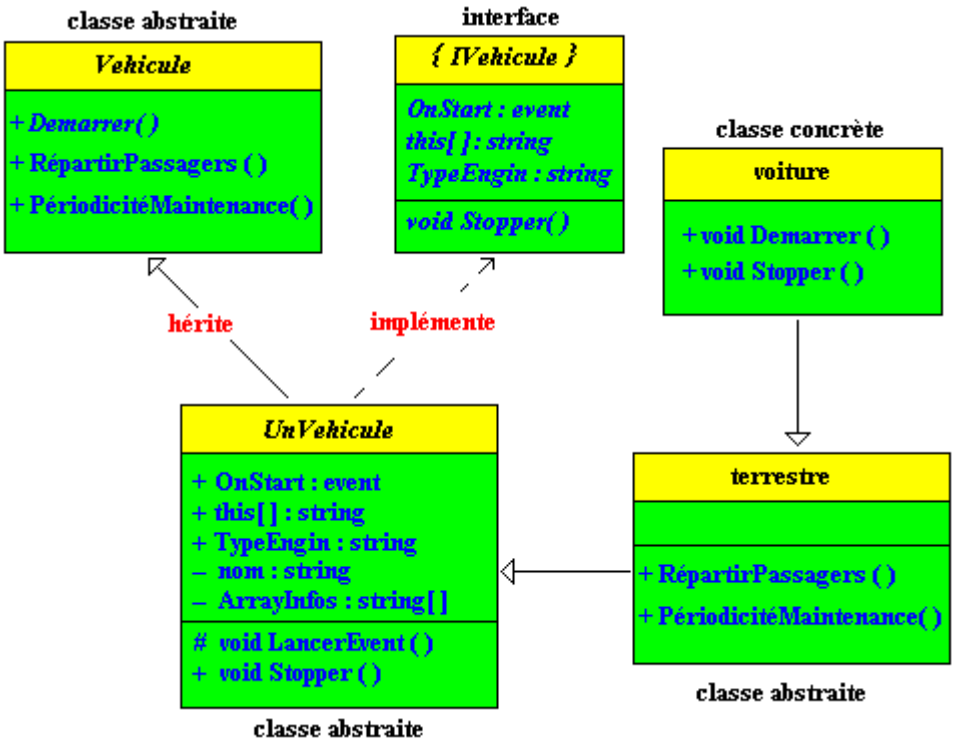


1.1.E) Une classe concrète

Nous finissons notre hiérarchie par une classe **Voiture** qui descend de la classe **Terrestre ,** qui implante la méthode **Demarrer**( ) et qui redéfinie la méthode **Stopper**( ) :



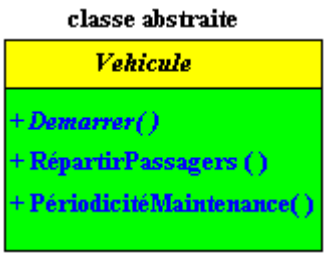
Ce qui nous donne le schéma d'héritage total suivant :



1.2 Implantation en C# de l'exemple

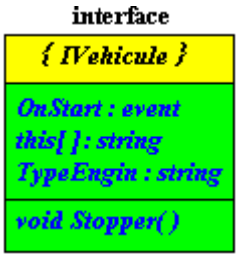
Nous proposons ci-dessous pour chaque classe ou interface une implémenation en C#.

1.2.A) La classe abstraite Vehicule



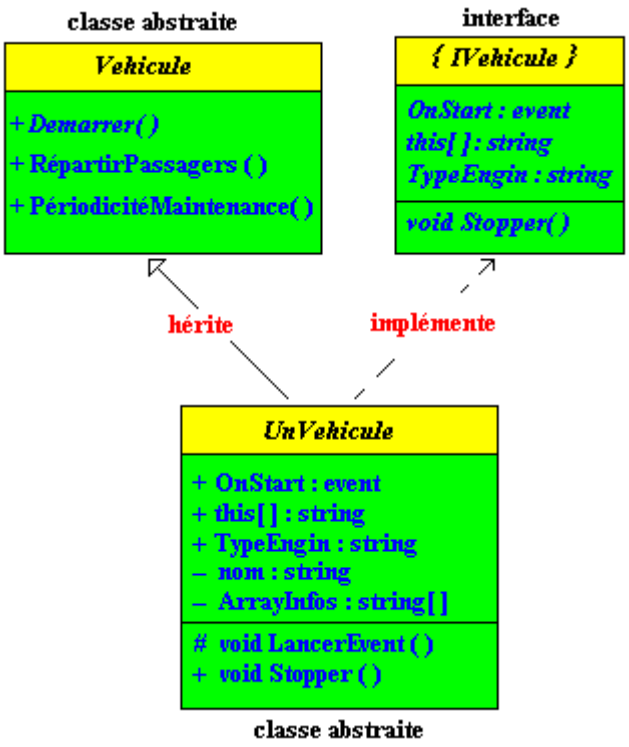

```
abstract class  Vehicule  // classe abstraite mère
{
   public abstract void  Demarrer ( ); // méthode abstraite
   public void  RépartirPassagers ( )  {  } // implantation de méthode avec corps vide
   public void  PériodicitéMaintenance ( ) {  } // implantation de méthode avec corps vide
}
```

1.2.B) L'interface IVehicule



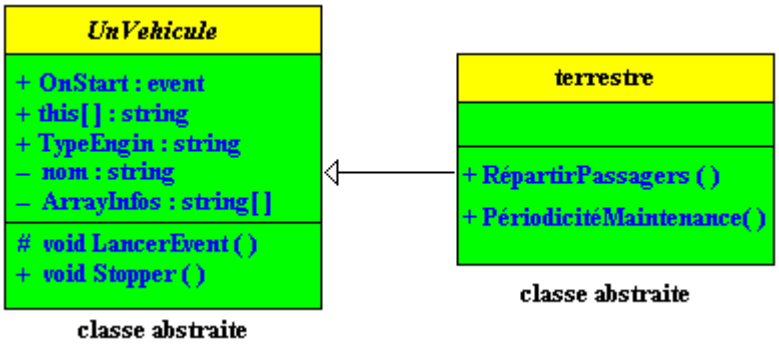

```
public delegate void  Starting ( );    //  declaration de type délégué
interface  IVehicule
{
   event  Starting OnStart ;   // déclaration d'événement du type délégué : Starting
    string  this  [ int  index]    // déclaration d'indexeur
    {
       get ;
       set ;
    }
    string  TypeEngin    // déclaration de propriété
    {
       get ;
       set ;
    }
   void  Stopper ( ); // déclaration de méthode
}
```

1.2.C) La classe UnVehicule



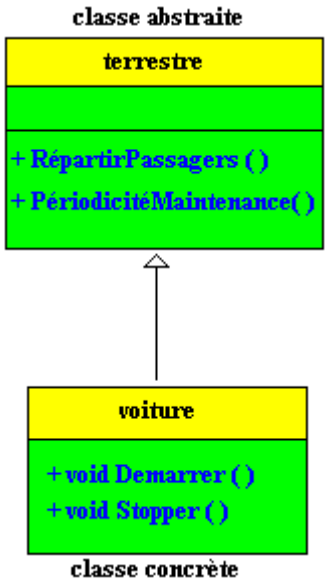

```
abstract class  UnVehicule : Vehicule , IVehicule  // hérite de la classe mère et implémente l'interface
{
  private  string  nom = "";
  private  string [ ] ArrayInfos = new  string [10] ;
  public  event  Starting OnStart ;
  protected void  LancerEvent ()
  {
    if( OnStart  !=  null)
       OnStart ();
  }
  public  string  this  [ int  index]    // implantation Indexeur
   {
     get      {       return  ArrayInfos[index] ;      }
     set      {       ArrayInfos[index] = value ;      }
  }
  public  string  TypeEngin    // implantation propriété
   {
     get      {       return  nom ;      }
     set      {       nom = value ;      }
  }
  public virtual void  Stopper ( )  {  } // implantation de méthode avec corps vide
}
```

1.2.D) La classe Terrestre




```
abstract class  Terrestre  :  UnVehicule
{
  public new void  RépartirPassagers ( )    {
    //...implantation de méthode
  }
  public new void  PériodicitéMaintenance ( )  {
   //...implantation de méthode
  }
}
```

1.2.E) La classe Voiture




```
class  Voiture  :  Terrestre
{
  public override void   Demarrer ( )  {
     LancerEvent ( );
  }
  public override void  Stopper ( ) {
     //...
   }
 }
```

2. Analyse du code de liaison de la solution précédente

Nous nous interessons au mode de liaison des membres du genre :

méthodes, **propriétés**, **indexeurs** et **événements**.

Rappelons au lecteur que la liaison statique indique que le compilateur lie le code lors de la compilation, alors que dans le cas d'une liaison dynamique le code n'est choisi et lié que lors de l'exécution.

2.1 Le code de la classe Vehicule

```
abstract class  Vehicule
{
  public abstract void  Demarrer ( ); // méthode abstraite
  public void  RépartirPassagers ( )  {  } // implantation de méthode avec corps vide
  public void  PériodicitéMaintenance ( ) {  } // implantation de méthode avec corps vide
}
```

Analyse :

Sans qualification particulière une méthode est à liaison statique :

- La méthode **public void** RépartirPassagers **( )** est donc à laison **statique**.
- La méthode **public void** PériodicitéMaintenance **( )** est donc à laison **statique**.

Une méthode qualifiée **abstract** est implicitement virtuelle :

- La méthode **public abstract void** Demarrer **( )** est donc à laison **dynamique**.

2.2 Le code de l'interface IVehicule

```
interface  IVehicule
{
  event  Starting OnStart ;   // déclaration d'événement du type délégué : Starting
   string  this  [ int  index]    // déclaration d'indexeur
   {      get ;      set ;   }
   string  TypeEngin    // déclaration de propriété
   {      get ;      set ;   }
  void  Stopper ( ); // déclaration de méthode
}
```

Analyse :

Une interface n'est qu'un contrat, les membres déclarés comme signatures dans l'interface n'étant pas implémentées, la question de leur liaison ne se pose pas au niveau de l'interface, mais lors de l'implémentation dans une classe ultérieure :

- La méthode **void** Stopper **( );** pourra donc être plus tard soit statique, soit dynamique.
- L'événement **event** Starting OnStart **;** pourra donc être plus tard soit statique, soit dynamique.
- La propriété string TypeEngin , pourra donc être plus tard soit statique, soit dynamique.
- L'indexeur string **this** [int index] , pourra donc être plus tard soit statique, soit dynamique.

2.3 Le code de la classe UnVehicule

```
abstract class UnVehicule : Vehicule , IVehicule
{
  private string nom = "";
  private string [ ] ArrayInfos = new string [10] ;
  public event Starting OnStart ;
  protected void LancerEvent ( )  {
    if( OnStart != null)  OnStart ();
  }
  public string this [ int index]   // implantation Indexeur
  {
    get {  return ArrayInfos[index] ; }
    set {  ArrayInfos[index] = value ;  }
  }
  public string TypeEngin   // implantation propriété
  {
    get { return nom ; }
    set { nom = value ; }
  }
  public virtual void Stopper ( )  { } // implantation de méthode avec corps vide
}
```

Analyse :

Le qualificateur **virtual** indique que l'élément qualifié est virtuel, donc à liaison dynamique; sans autre qualification un élément est par défaut à liaison statique :

- La méthode **public virtual void** Stopper **( )** est à liaison **dynamique**.
- L'événement **public event** Starting OnStart est à liaison **statique**.
- La propriété **public** string TypeEngin est à liaison **statique**.

- L'indexeur **public** string **this** [int index] est à liaison **statique**.
- La méthode **protected void** LancerEvent **( )** est à liaison **statique**.

2.4 Le code de la classe Terrestre

```
abstract class Terrestre : UnVehicule
{
  public new void RépartirPassagers ( )  {
    //...implantation de méthode
  }
  public new void PériodicitéMaintenance ( ) {
   //...implantation de méthode
  }
}
```

Analyse :

Dans la classe mère Vehicule les deux méthodes RépartirPassagers et PériodicitéMaintenance sont à liaison statique, dans la classe Terrestre :

- La méthode **public new void** RépartirPassagers **( )** est à laison **statique** et masque la méthode mère.
- La méthode **public new void** PériodicitéMaintenance **( )** est à laison **statique** et masque la méthode mère.

2.5 Le code de la classe Voiture

```
class Voiture : Terrestre
{
  public override void  Demarrer ( ) {
     LancerEvent ( );
  }
  public override void Stopper ( ) {
     //...
   }
 }
```

Analyse :

La méthode Demarrer( ) est héritée de la classe mère Vehicule, la méthode Stopper( ) est héritée de la classe UnVehicule :

- La méthode **public override void** Demarrer **( )** est à laison **dynamique** et redéfinit la méthode abstraite mère.
- La méthode **public override void** Stopper **( )** est à laison **dynamique** et redéfinit la méthode virtuelle à corps vide de la classe UnVehicule.

3. Cohérence entre les notions de classe et d'interface dans C#

Une classe peut implémenter plusieurs interfaces. Dans ce cas nous avons une excellente alternative à l'héritage multiple.

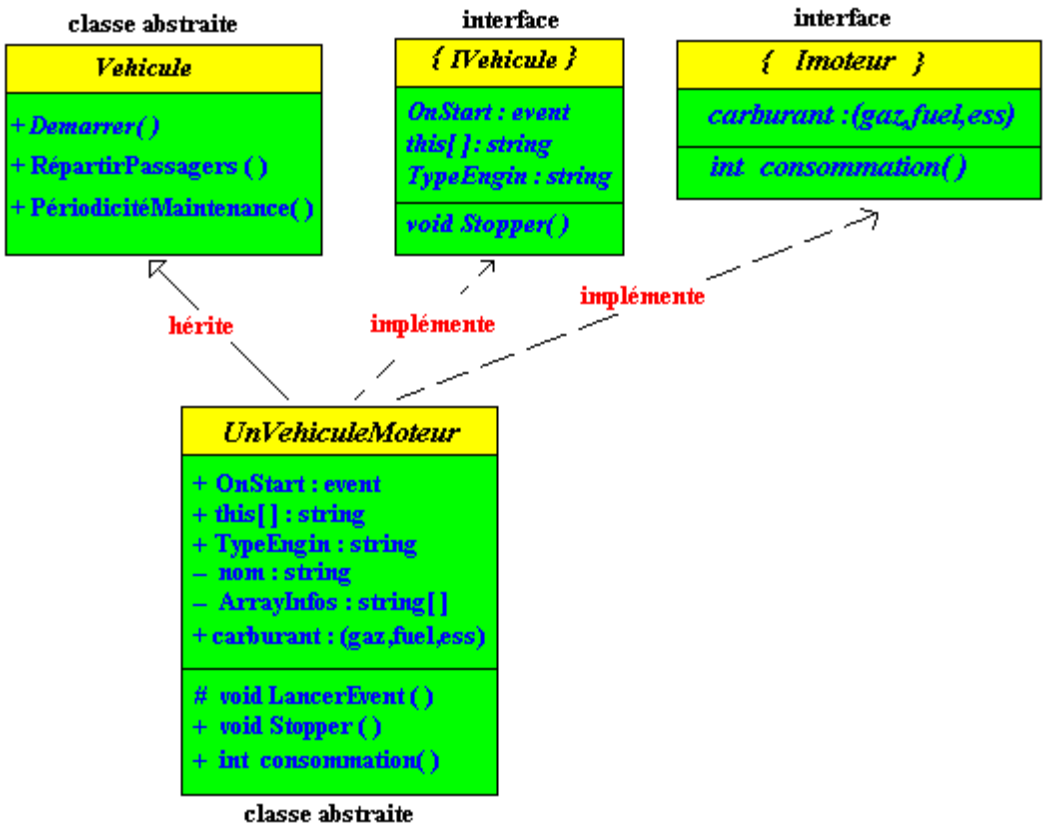
Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui ne devraient plus être modifiés. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

Les variables de types interface respectent les mêmes règles de **transtypage** que les variables de types classe.

Les **objets** de type classe **clA** peuvent être transtypés et **referencés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

Une classe peut implémenter plusieures interfaces

Soit la définition suivante où la classe UnVehiculeMoteur hérite de la classe abstraite Vehicule et implémente l'interface IVehicule de l'exemple précédent, et supposons qu'en plus cette classe implémente l'interface IMoteur. L'interface IMoteur explique que lorsqu'un véhicule est à moteur, il faut se préoccuper de son type de carburant et de sa consommation :



Ci-dessous le code C# correspondant à cette définition, plus une classe concrète **Voiture** instanciable dérivant de la classe abstraite UnVehiculeMoteur :

```
abstract class Vehicule {
  public abstract void Demarrer ( ); // méthode abstraite
  public void RépartirPassagers ( ) { } // implantation de méthode avec corps vide
  public void PériodicitéMaintenance ( ) { } // implantation de méthode avec corps vide
}
interface IVehicule {
  event Starting OnStart ; // déclaration d'événement du type délégué : Starting
   string this [ int index] // déclaration d'indexeur
   { get ; set ; }
   string TypeEngin // déclaration de propriété
   { get ; set ; }
  void Stopper ( ); // déclaration de méthode
}
enum Energie { gaz , fuel , ess } // type énuméré pour le carburant

interface IMoteur {
    Energie carburant // déclaration de propriété
     { get ; }
    int consommation ( ); // déclaration de méthode
}
```

```
abstract class UnVehiculeMoteur : Vehicule , IVehicule , IMoteur
{
  private string nom = "";
  private Energie typeEnerg = Energie.fuel ;
  private string [ ] ArrayInfos = new string [10] ;
  public event Starting OnStart ;
```

```
  protected void  LancerEvent ( )   {
    if( OnStart  !=  null)   OnStart ( );
  }
  public  string  this  [ int  index]    // implantation Indexeur de IVehicule
  {
     get  {   return  ArrayInfos[index] ;  }
     set  {   ArrayInfos[index] = value ;   }
  }
  public  string  TypeEngin    // implantation propriété de IVehicule
  {
     get  {  return  nom ;  }
     set  {  nom = value ;  }
  }
       public  Energie  carburant    // implantation propriété de IMoteur
       {
          get  {  return  typeEnerg ;  }
       }
       public virtual void  Stopper ( )   {  } // implantation vide de méthode  de IVehicule
       public virtual int  consommation ( )   { return .... } // implantation de méthode  de IMoteur
     }
```

```
class  Voiture  :  UnVehiculeMoteur  {
  public override void   Demarrer ( )  {
     LancerEvent ( );
  }
  public override void  Stopper ( ) {
    //...implantation de méthode
   }
 public new void  RépartirPassagers ( )    {
    //...implantation de méthode
  }
  public new void  PériodicitéMaintenance ( )  {
   //...implantation de méthode
  }
}
```

Les interfaces et classes respectent les mêmes règles de polymorphisme

Il est tout à fait possible d'utiliser des variables de référence sur des interfaces et de les transtyper d'une manière identique à des variables de référence de classe. En particulier le polymorphisme de référence s'applique aux références d'interfaces.

Le polymorphisme de référence sur les classes de l'exemple précédent	

abstract class Vehicule **{ ... }**
interface IVehicule **{ ... }**
enum Energie **{** gaz , fuel , ess **}**
interface IMoteur **{ ... }**
 	
abstract class UnVehiculeMoteur **:** Vehicule , IVehicule , IMoteur **{ ... }**
class Voiture **:** UnVehiculeMoteur **{ ... }**
 	
class UseVoiture1 **{**
public string use (**IVehicule** x){
return x.TypeEngin ;
}
	class MaClass **{**
	
	static void Main(string [ ] args) **{**
	const string ch; s = "le client utilise une ";
	string ch ;
	IVehicule a1;
	UnVehiculeMoteur b1;
	Voiture c1;
	a1 = **new** Voiture( );
	a1.TypeEngin = "Renault";
	b1= **new** Voiture( );
	b1.TypeEngin = "Citroën";

```
}
class UseVoiture2 {
  public string use ( UnVehiculeMoteur x ){
      return x.TypeEngin ;
  }
}
class UseVoiture3 {
  public string use ( Voiture x ){
      return x.TypeEngin ;
  }
}
```

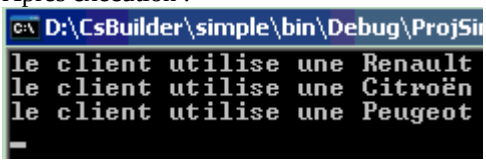
```
  c1 = new Voiture( );
  c1.TypeEngin = "Peugeot";
  UseVoiture1 client = new UseVoiture1( );
  ch = s+client.use(a1);
  System.Console.WriteLine(ch);
  ch = s+client.use(b1);
  System.Console.WriteLine(ch);
  ch = s+client.use(c1);
  System.Console.WriteLine(ch);
 }
}
```

code d'exécution avec 3 objets différents

```
static void Main(string [ ] args)  {
 .... idem
UseVoiture1 client = new UseVoiture1( );
 ch = s+client.use(a1);
 System.Console.WriteLine(ch);
 ch = s+client.use(b1);
 System.Console.WriteLine(ch);
 ch = s+client.use(c1);
 System.Console.WriteLine(ch);
}
```

IVehicule
|__UnVehiculeMoteur
|__Voiture

Après exécution :



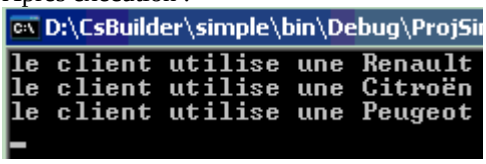
a1 est une référence sur IVehicule, b1 est une référence sur une interface fille de IVehicule, c1 est de classe Voiture implémentant IVehicule.

Donc chacun de ces trois genres de paramètre peut être passé par polymorphisme de référence d'objet à la méthode **public string** use (**IVehicule** x)

```
static void Main(string [ ] args)  {
.... idem
UseVoiture2 client = new UseVoiture2( );
 ch = s+client.use((UnVehiculeMoteur)a1);
 System.Console.WriteLine(ch);
 ch = s+client.use(b1);
 System.Console.WriteLine(ch);
 ch = s+client.use(c1);
 System.Console.WriteLine(ch);
}
```

IVehicule
|__UnVehiculeMoteur
|__Voiture

Après exécution :



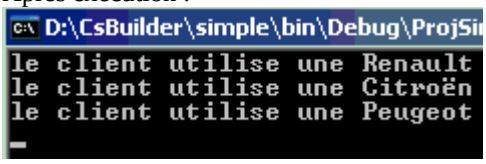
Le polymorphisme de référence d'objet appliqué à la méthode **public string** use (**UnVehiculeMoteur** x) indique que les paramètres passés doivent être de type UnVehiculeMoteur ou de type descendant.

La variable a1 est une référence sur IVehicule qui ne descend pas de UnVehiculeMoteur, il faut donc transtyper la référence a1 soit : (**UnVehiculeMoteur**)a1.

```
static void Main(string [ ] args)  {
.... idem
UseVoiture3 client = new UseVoiture3( );
 ch = s+client.use( (Voiture)a1 );
 System.Console.WriteLine(ch);
 ch = s+client.use( (Voiture)b1 );
 System.Console.WriteLine(ch);
 ch = s+client.use(c1);
 System.Console.WriteLine(ch);
}
```

IVehicule
|__UnVehiculeMoteur

Après exécution :



Le polymorphisme de référence d'objet appliqué à la méthode **public string** use (**Voiture** x) indique que les paramètres passé doivent être de type Voiture ou de type descendant.

La variable a1 est une référence sur IVehicule qui ne descend pas de Voiture, il faut donc transtyper la référence a1 soit :

|__**Voiture**	(**Voiture**)a1 La variable b1 est une référence sur UnVehiculeMoteur qui ne descend pas de Voiture, il faut donc transtyper la référence b1 soit : (**Voiture**)b1

Les opérateurs **is** et **as** sont utilisables avec des références d'interfaces en C#. Reprenons l'exemple précédent :

```
abstract class  Vehicule   { ... }
interface  IVehicule { ... }
enum Energie { gaz , fuel , ess }
interface IMoteur { ... }
abstract class  UnVehiculeMoteur : Vehicule , IVehicule , IMoteur { ... }
class  Voiture : UnVehiculeMoteur  { ... }

class UseVoiture1 {
  public string use ( IVehicule x ){
      if (x is UnVehiculeMoteur) {
        int consom =   (x as UnVehiculeMoteur).consommation( );
        return " consommation="+consom.ToString( ) ;
      }
     else
       return x.TypeEngin ;
  }
}
```

Les conflits de noms dans les interfaces

Il est possible que deux interfaces différentes possèdent des membres ayant la même signature. Une classe qui implémente ces deux interfaces se trouvera confrontée à un conflit de nom (ambiguïté). Le compilateur C# exige dès lors que l'ambiguïté soit levée avec le préfixage du nom du membre par celui de l'interface correspondante (implémentation explicite).

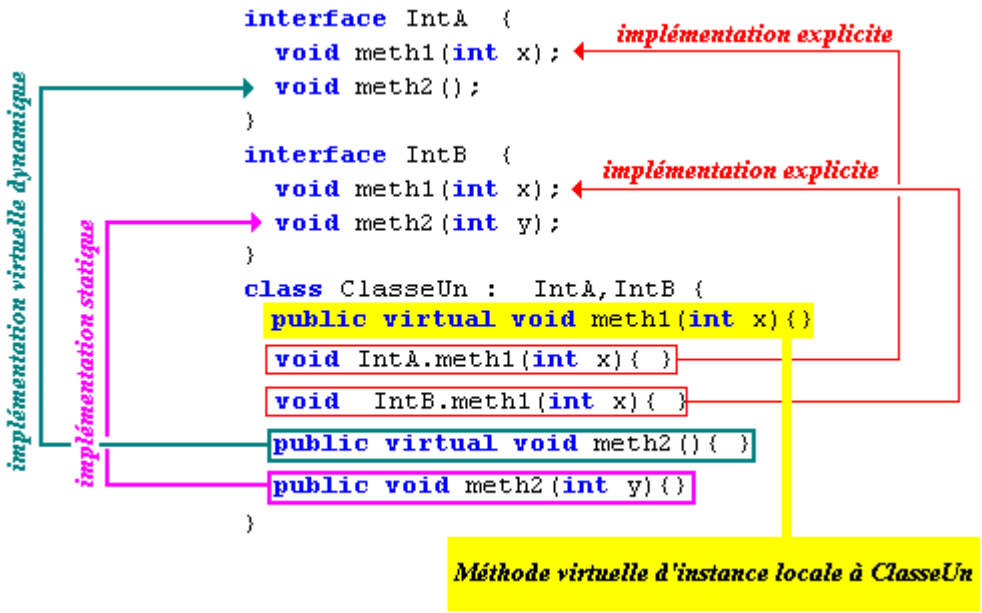
L'exemple ci-dessous est figuré avec deux interfaces IntA et IntB contenant chacune deux méthodes portant les mêmes noms et plus particulièrement la méthode meth1 possède la même signature dans chaque interface. Soit ClasseUn une classe implémentant ces deux interfaces. Voici comment fait C# pour choisir les appels de méthodes implantées.

Le code source de ClasseUn implémentant les deux interfaces IntA et IntB :

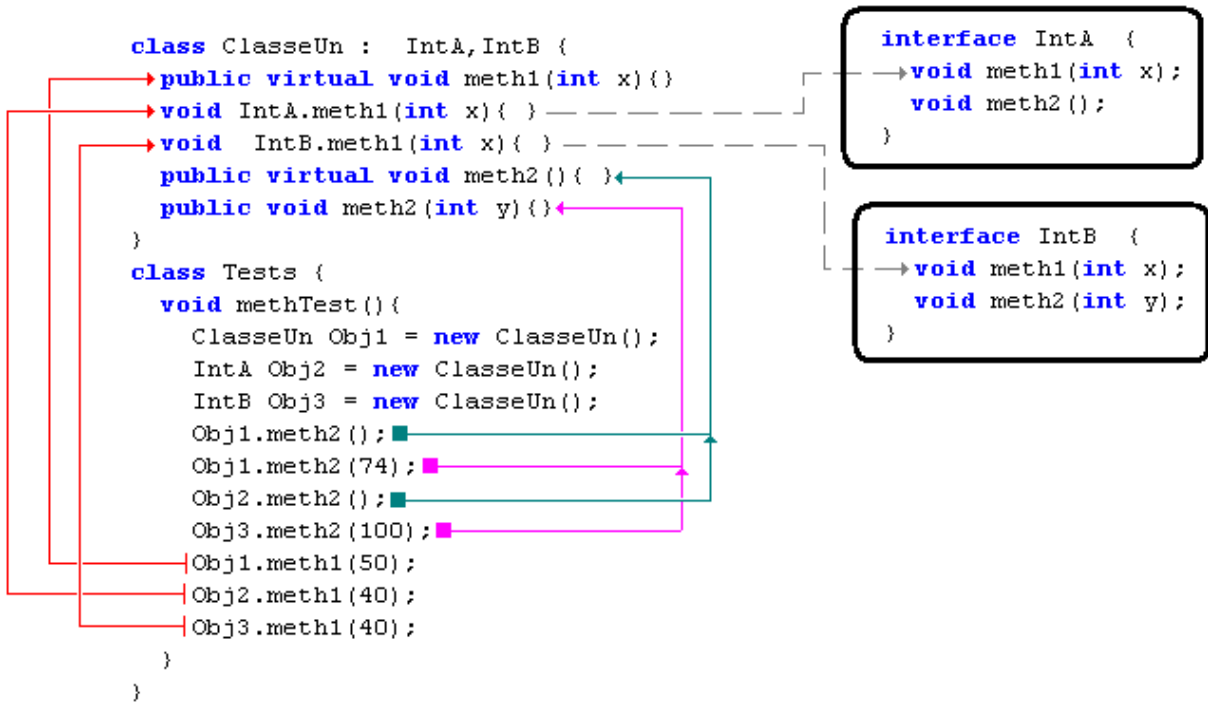
```
interface IntA{
 void meth1( int x );
 void meth2( );
}
interface IntB{
 void meth1( int x );
 void meth2( int y );
}
```

```
class ClasseUn :  IntA , IntB {
   public virtual void meth1( int x ){ }
   void IntA.meth1( int x ){ }
   void IntB.meth1( int x ){ }
   public virtual void meth2( ){ }
   public void meth2( int y ){ }
}
```

Schéma expliquant ce que C# analyse dans le code source précédent :



Lorsque l'on instancie effectivement 3 objets de classe **ClasseUn** précédente, et que l'on déclare chaque objet avec un type de référence différent : classe ou interface, le schéma ci-dessous indique quels sont les différents appels de méthodes corrects possibles :



Il est aussi possible de transtyper une référence d'objet de classe ClasseUn en une référence d'interface dont elle hérite (les appels sont identiques à ceux du schéma précédent) :

```
class Tests    {
      void methTest( ) {
         ClasseUn Obj1 = new ClasseUn( );
         IntA Obj2 = ( IntA )Obj1;
```

```
        IntB Obj3 = ( IntB )Obj1;
        Obj1.meth2( );
        Obj1.meth2(74);
        Obj2.meth2( );
        Obj3.meth2(100);
        Obj1.meth1(50);
        Obj2.meth1(40);
        Obj3.meth1(40);
    }
}
```

Nous remarquons qu'aucun conflit et aucune ambiguïté de méthode ne sont possibles et que grâce à l'implémentation explicite, toutes les méthodes de même nom sont accessibles.

Enfin, nous avons préféré utiliser le transtypage détaillé dans :

```
IntA Obj2 = ( IntA )Obj1 ;
Obj2.meth1(40) ;
```

plutôt que l'écriture équivalente :

```
(( IntA )Obj1).meth1(40) ; //...appel de IntA.meth1(int x)
```

Car l'oubli du parenthésage externe dans l'instruction " ((IntA)Obj1).meth1(40) " peut provoquer des incompréhensions dans la mesure où aucune erreur n'est signalé par le compilateur car ce n'est plus la même méthode qui est appelée.

```
( IntA )Obj1.meth1(40) ; //...appel de public virtual ClasseUn.meth1(int x)
```

Classe de délégation

Plan général:

Les classes de délégations

1.1 Définition classe de délégation - délégué
1.2 Délégué et méthodes de classe - définition
1.3 Délégué et méthodes de classe - informations pendant l'exécution
1.4 Délégué et méthodes d'instance - définition
1.5 Plusieurs méthodes pour le même délégué
1.6 Exécution des méthodes d'un délégué multicast - Exemple de code

Il existe en Delphi la notion de **pointeur de méthode** utilisée pour implanter la notion de gestionnaire d'événements. C# a repris cette idée en l'encapsulant dans un concept objet plus abstrait : la notion de **classes de délégations**. Dans la machine virtuelle CLR, la gestion des événements est fondée sur les classes de délégations, il est donc essentiel de comprendre le modèle du délégué pour comprendre le fonctionnement des événements en C#.

1. Les classes de délégations

Note de microsoft à l'attention des développeurs :

> *La classe **Delegate** est la classe de base pour les types délégué. Toutefois, seuls le système et les compilateurs peuvent dériver de manière explicite de la classe **Delegate** ou **MulticastDelegate**. En outre, **il n'est pas possible de dériver un nouveau type d'un type délégué**. La classe **Delegate** n'est pas considérée comme un type délégué. Il s'agit d'une classe utilisée pour dériver des types délégué.*

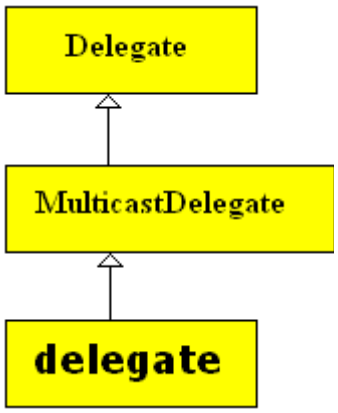
1.1 Définition classe de délégation - délégué

Le langage C# contient un mot clé **delegate** , permettant au compilateur de construire une classe dérivée de la classe **MulticastDelegate** dérivée elle-même de la classe **Delegate**. Nous ne pouvons pas instancier une objet de classe **Delegate**, car le constructeur est spécifié **protected** et n'est donc pas accessible :

Constructeurs protégés

Delegate, constructeur	Surchargé. Initialise un nouveau délégué.

C'est en fait via ce mot clé **delegate** que nous allons construire des classes qui ont pour nom : classes de délégations. Ces classes sont des classes du genre référence, elles sont instanciables et un objet de classe délégation est appelé un délégué.



Un objet de classe délégation permet de référencer (pointer vers) une ou plusieurs méthodes.

Il s'agit donc de l'extension de la notion de pointeur de méthode de Delphi. Selon les auteurs une classe de délégation peut être ausi nommée classe déléguée, type délégué voir même tout simplement délégué. Il est essentiel dans le texte lu de bien distinguer la classe et l'objet instancié.

Lorsque nous utiliserons le vocable classe délégué ou type délégué nous parlerons de la classe de délégation d'un objet délégué.

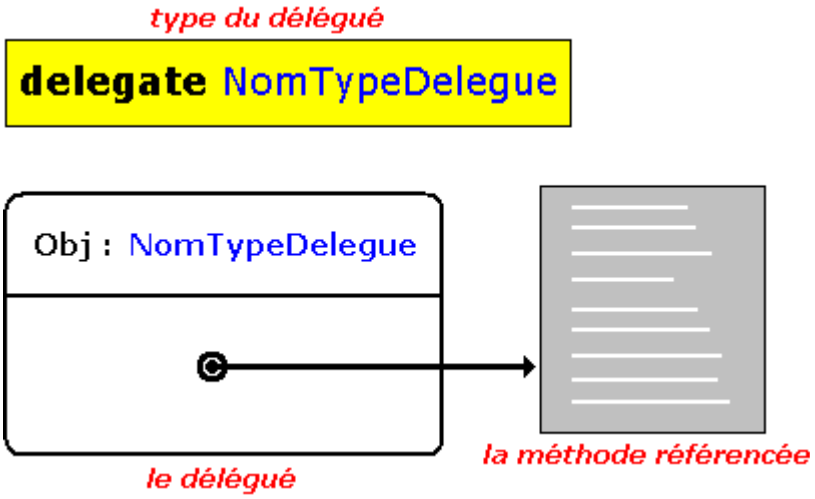
Il est donc possible de créer un nouveau type de délégué (une nouvelle classe) dans un programme, ceci d'une seule manière : en utilisant le qualificateur **delegate**. Ci-dessous nous déclarons un type délégué (une nouvelle classe particulière) nommé NomTypeDelegue :

```
delegate string  NomTypeDelegue ( int parametre) ;
```

Un objet délégué peut donc être instancé à partir de cette "classe" comme n'importe quel autre objet :

```
NomTypeDelegue Obj = new NomTypeDelegue ( <paramètre> )
```

Il ne doit y avoir qu'**un seul paramètre** <paramètre> et c'est obligatoirement un **nom de méthode**. Soit MethodeXYZ le nom de la méthode passé en paramètre, nous dirons alors que le délégué (l'objet délégué) référence la méthode MethodeXYZ .



Les méthodes référencées par un délégué peuvent être :

des méthodes de classe (**static**)
ou
des méthodes d'instance

Toutes les méthodes référencées par un même délégué ont la même signature partielle :

- même type de retour du résultat,
- même nombre de paramètres,
- même ordre et type des paramètres,
- seul leur nom diffère.

1.2 Délégué et méthodes de classe - définition

Un type commençant par le mot clef **delegate** est une classe délégation.

ci-dessous la syntaxe de 2 exemples de déclaration de classe délégation :

- **delegate string** Deleguer1(**int** x) ;
- **delegate void** Deleguer2(**string** s) ;

Un objet instancié à partir de la classe Deleguer1 est appelé un délégué de classe Deleguer1 :

- Deleguer1 FoncDeleg1 = **new** Deleguer1 (Fonc1) ;
- où Fonc1 est une méthode :
- **static string** Fonc1 (**int** x) { ... }

Un objet instancié à partir de la classe Deleguer2 est appelé un délégué de classe Deleguer2 :

- Deleguer2 FoncDeleg2 = **new** Deleguer2 (Fonc2) ;
- où Fonc2 est une autre méthode :
- **static void** Fonc2 (**string** x) { ... }

Nous avons créé deux types délégations nommés Deleguer1 et Deleguer2 :

- Le type Deleguer1 permet de référencer des méthodes ayant un paramètre de type **int** et renvoyant un **string**.
- Le type Deleguer2 permet de référencer des méthodes ayant un paramètre de type **string** et ne renvoyant **rien**.

Les fonctions de classe Fonc1 et Fonc11 répondent à la signature partielle du type Deleguer1

- **static string** Fonc1 (**int** x) { ... }
- **static string** Fonc11 (**int** x) { ... }

On peut créer un objet (un délégué) qui va référencer l'une ou l'autre de ces deux fonctions :

- Deleguer1 FoncDeleg1 = **new** Deleguer1 (Fonc1) ;
  ou bien
- Deleguer1 FoncDeleg1 = **new** Deleguer1 (Fonc11) ;

On peut maintenant appeler le délégué FoncDeleg1 dans une instruction avec un paramètre d'entrée de type **int**, selon que le délégué référence Fonc1 ou bien Fonc11 c'est l'une ou l'autre des ces fonctions qui est en fait appelée.

Source d'un exemple C# et exécution :

```
delegate string Deleguer1 ( int x );

class ClasseA {
 static string Fonc1 ( int x ) {
  return ( x * 10 ) .ToString ( );
 }
 static string Fonc11 ( int x ) {
  return ( x * 100 ) .ToString ( );
 }
 static void Main ( string [] args ) {
   string s = Fonc1 ( 32 );
   System .Console.WriteLine ("Fonc1(32) = " + s );
   s = Fonc11 ( 32 ); // appel de fonction classique
   System .Console.WriteLine ("Fonc11(32) = " + s );
   System .Console.WriteLine ("\nLe délégué référence Fonc1 :");
   Deleguer1 FoncDeleg1 = new Deleguer1 ( Fonc1 ) ;
```

```
    s  =  FoncDeleg1 ( 32 ); // appel au délégué qui appelle la fonction
    System .Console.WriteLine ("FoncDeleg1(32) = " + s );
    System .Console.WriteLine ("\nLe délégué référence maintenant Fonc11 :");
    FoncDeleg1 = new  Deleguer1  ( Fonc11 ) ; // on change d'objet référencé (de fonction)
    s  =  FoncDeleg1 ( 32 ); // appel au délégué qui appelle la fonction
    System .Console.WriteLine ("FoncDeleg1(32) = " + s );
    System .Console.ReadLine ( );
  }
}
```

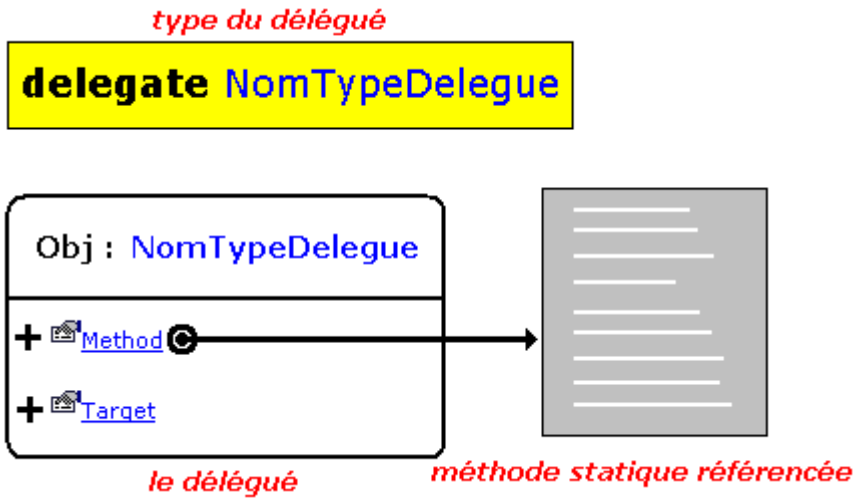
Résultats d'exécution sur la console :

```
D:\CsBuilder\Delegates\bin\Debug\PrDelegate.exe
Fonc1(32) = 320
Fonc11(32) = 3200

Le délégué référence Fonc1 :
FoncDeleg1(32) = 320

Le délégué référence maintenant Fonc11 :
FoncDeleg1(32) = 3200
```

1.3 Délégué et méthodes de classe - informations pendant l'exécution

Comme une référence de délégué peut pointer (référencer) vers des méthodes de classes différentes au cours de l'exécution, il est intéressant d'obtenir des informations sur la méthode de classe actuellement référencée par le délégué.

Nous donnons ci-dessous les deux propriétés publiques qui sont utiles lors de cette recherche d'informations, elles proviennent de la classe mère Delegate non héritable par programme :



Propriétés publiques

Method (hérité de **Delegate**)	Obtient la méthode static représentée par le délégué.
Target (hérité de **Delegate**)	Obtient l'instance de classe sur laquelle le délégué en cours appelle la méthode d'instance.

La propriété **Target** sert plus particulièrement lorsque le délégué référence une méthode d'instance, la propriété **Method** est la seule utilisable lorsque la méthode référencée par le délégué est une méthode de classe (méthode marquée **static**). Lorsque la méthode référencée par le délégué est une méthode de classe le champ **Target** a la valeur **null**.

Ci-dessous nous avons extrait quelques informations concernant la propriété **Method** qui est elle-même de classe MethodInfo :

Method (hérité de **Delegate**)

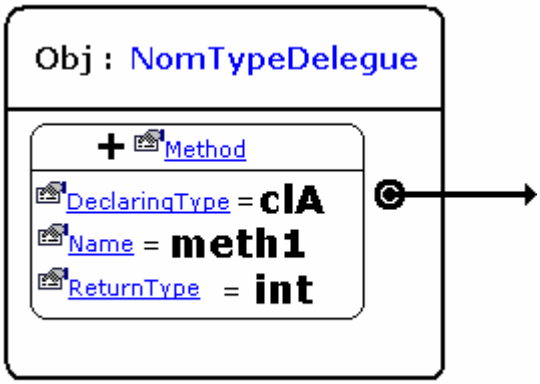
Obtient la méthode static représentée par le délégué.

```
[Serializable]
[ClassInterface(ClassInterfaceType.AutoDual)]
public MethodInfo Method {get;}
```

MethodInfo : Propriétés publiques

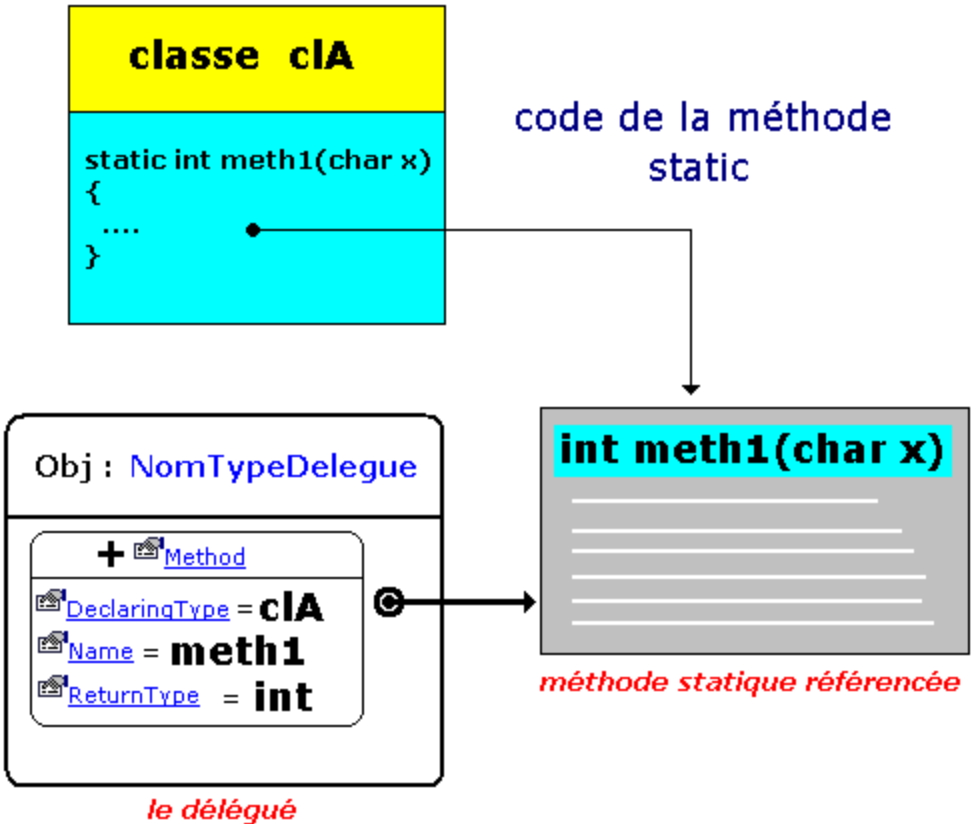
CallingConvention (hérité de **MethodBase**)	Obtient une valeur indiquant les conventions d'appel de cette méthode.
DeclaringType (hérité de **MemberInfo**)	Obtient la classe qui déclare ce membre.
Name (hérité de **MemberInfo**)	Obtient le nom de ce membre.
ReturnType	Obtient le type de retour de cette méthode.

Ces propriétés sont des membres de la propriété **Method** qui est applicable uniquement lorsque le délégué en cours référence une méthode de classe (qualifiée **static**).



le délégué

Nous illustrons dans la figure ci-après, dans le cas d'une méthode de classe, l'utilisation des propriétés **Name**, **DeclaringType** et **ReturnType** membres de la propriété **Method** :



Nous obtenons ainsi des informations sur le **nom**, le **type** du résultat et la **classe** de la méthode **static** pointée par le délégué.

Source complet exécutable d'un exemple d'information sur la méthode de classe référencée :

```
namespace PrDelegate {

 delegate  string  Deleguer1 ( int  x );

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

  static void  Main ( string [ ] args )  {
    System .Console.WriteLine ("\nLe délégué référence Fonc1 :");
    Deleguer1  FoncDeleg1  =  new  Deleguer1  (  Fonc1  ) ;
    System .Console.WriteLine ( "nom : "+FoncDeleg1.Method.Name );
    System .Console.WriteLine ("classe : "+ FoncDeleg1.Method.DeclaringType.ToString( ) );
    System .Console.WriteLine ( "retour : "+FoncDeleg1.Method.ReturnType.ToString( ) );
    System .Console.ReadLine ( );
  }
 }
}
```

Résultats d'exécution sur la console :

```
D:\CsBuilder\Delegates\bin\Debug\PrD

Le délégué référence Fonc1 :
nom : Fonc1
classe : PrDelegate.ClasseA
retour : System.String
```

1.4 Délégué et méthodes d'instance - définition

Outre une méthode de classe, un délégué peut pointer aussi vers une méthode d'instance(une méthode d'un objet). Le fonctionnement (déclaration, instanciation, utilisation) est identique à celui du référencement d'une méthode de classe, avec syntaxiquement l'obligation, lors de l'instanciation du délégué, d'indiquer le nom de l'objet ainsi que le nom de la méthode **Obj.Methode** (similitude avec le pointeur de méthode en Delphi).

Ci-dessous la syntaxe d'un exemple de déclaration de classe délégation pour une méthode d'instance, nous devons :

1°) Déclarer ne classe contenant une méthode public

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

2°) Déclarer un type délégation

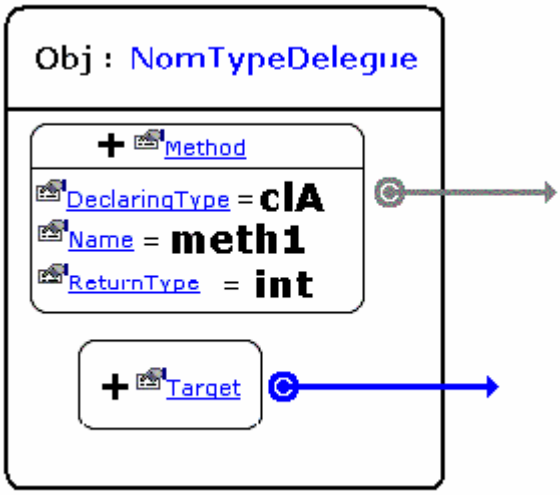
```
delegate int Deleguer( char x ) ;
```

3°) Instancier un objet de la classe clA

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

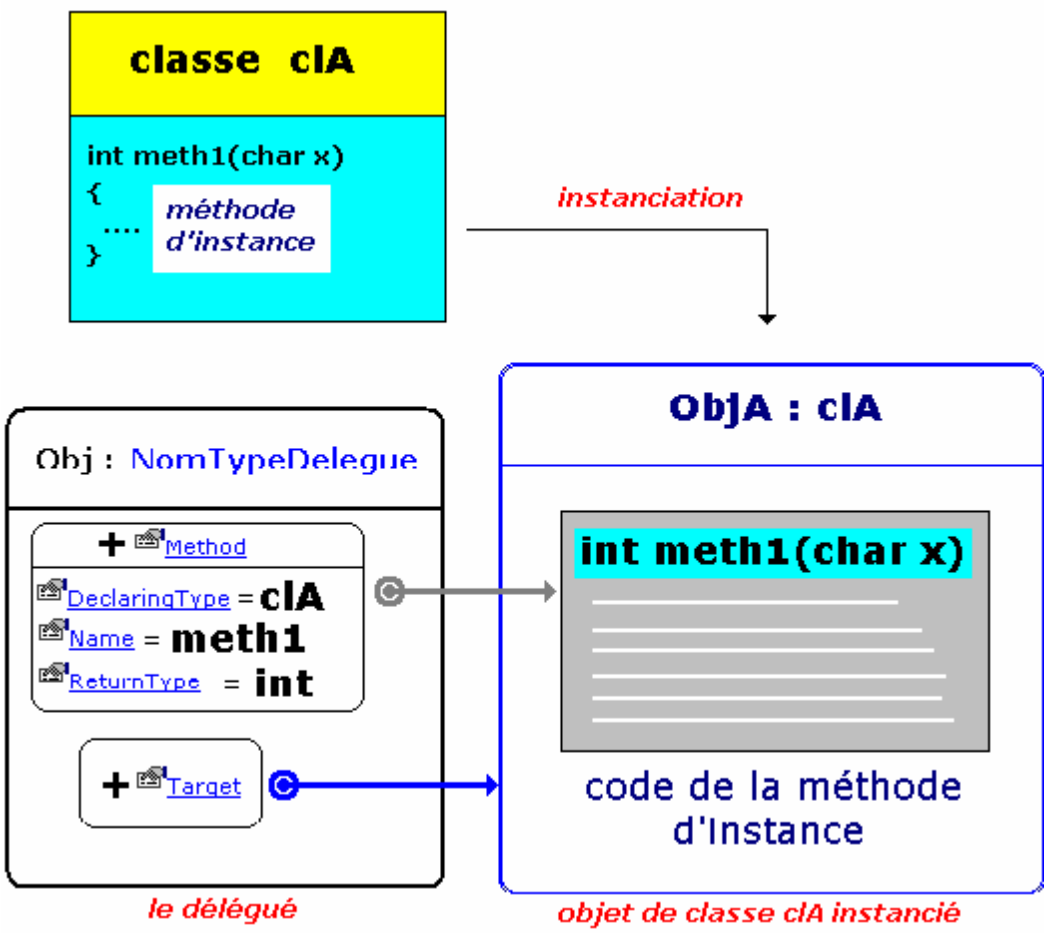
4°) Instancier à partir de la classe Deleguer un délégué

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



le délégué

Nous illustrons dans la figure ci-après, dans le cas d'une méthode d'instance, l'utilisation de membres de la propriété **Method** et de la propriété **Target** :



Source complet exécutable d'un exemple d'information sur la méthode d'instance référencée :

```
namespace PrDelegate {

 delegate  int  Deleguer ( char  x );

 class  ClasseA {
  public  int  meth1  (  char  x  )  {
    return   x  ;
  }

  static void  Main ( string [ ] args )  {
    ClasseA ObjX , ObjA = new ClasseA( );
    System .Console.WriteLine ("Un délégué référence ObjA.meth1 :");
    Deleguer  FoncDeleg  =  new  Deleguer  (  ObjA.meth1  ) ;
    ObjX = (ClasseA)FoncDeleg.Target;
    if (ObjX.Equals(ObjA))
          System.Console.WriteLine ("Target référence bien ObjA");
    else System.Console.WriteLine ("Target ne référence pas ObjA");
    System.Console.WriteLine ( "\nnom : "+FoncDeleg.Method.Name );
    System.Console.WriteLine ("classe : "+ FoncDeleg.Method.DeclaringType.ToString( ) );
    System.Console.WriteLine ( "retour : "+FoncDeleg.Method.ReturnType.ToString( ) );
    System.Console.ReadLine ( );
  }
 }
}
```

Résultats d'exécution sur la console :

```
D:\CsBuilder\Delegates\bin\Debug\PrDelegate.exe
Un délégué référence ObjA.meth1 :
Target référence bien ObjA

nom : meth1
classe : PrDelegate.ClasseA
retour : System.Int32
```

Dans le programme précédent, les lignes de code suivantes :

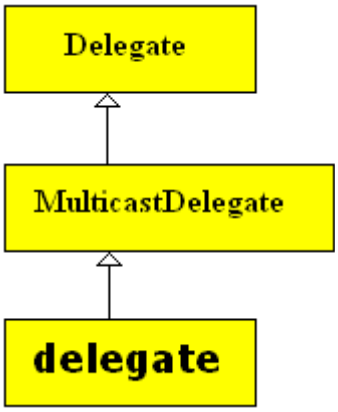
```
ObjX = (ClasseA)FoncDeleg.Target ;
if (ObjX.Equals(ObjA))
    System.Console.WriteLine ("Target référence bien ObjA") ;
else System.Console.WriteLine ("Target ne référence pas ObjA") ;
```

servent à faire "pointer" la référence *ObjX* vers l'objet vers lequel pointe *FoncDeleg.Target*. La référence de cet objet est transtypée car *ObjX* est de type *ClasseA*, *FoncDeleg.Target* est de type *Object* et le compilateur n'accepterait pas l'affectation *ObjX = FoncDeleg.Target*. Le test ***if*** *(ObjX.Equals(ObjA))*... permet de nous assurer que les deux références *ObjX* et *ObjA* pointent bien vers le même objet.

1.5 Plusieurs méthodes pour le même délégué

C# autorise le référencement de plusieurs méthodes par le même délégué, nous utiliserons le vocabulaire de délégué multicast pour bien préciser qu'il référence plusieurs méthodes. Le délégué multicast conserve les référencements dans une liste d'objet. Les méthodes ainsi référencées peuvent chacune **être du genre méthode de classe ou méthode d'instance**, elles **doivent avoir la même signature**.

Rappelons qu'un type délégué multicast est une classe qui hérite intrinsèquement de la classe **MulticasteDelegate** :



La documentation de .Net Framework indique que la classe MulticasteDelegate contient en particulier trois champs privés :

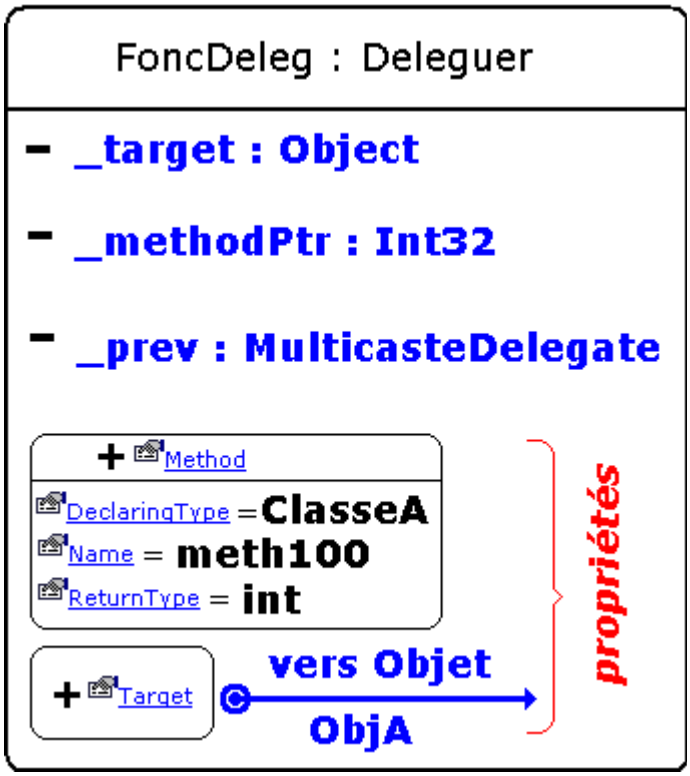
```
Object  _target ;
Int32  _methodPtr ;
```

MulticasteDelegate _prev ;
Le champ **_prev** est utilisé pour maintenir une liste de MulticasteDelegate

Lorsque nous déclarons un programme comme celui-ci :

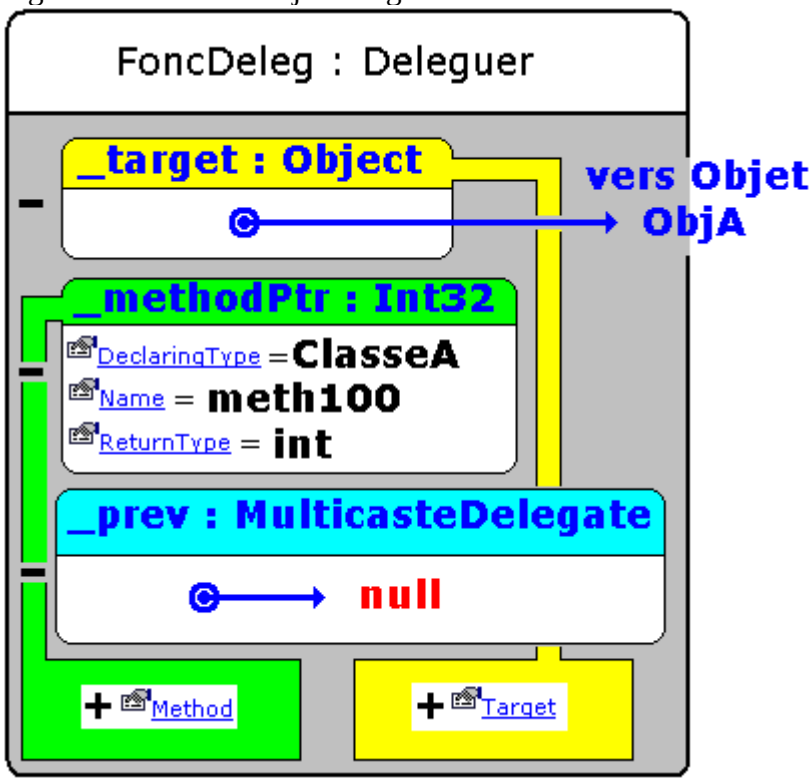
```
delegate int Deleguer ( char x );

class ClasseA {
 public int meth100 ( char x ) {
    System.Console.WriteLine ("Exécution de meth100('"+x+"')");
    return x+100 ;
 }
 static void Main ( string [ ] args ) {
   ClasseA ObjA = new ClasseA( );
   Deleguer FoncDeleg = new Deleguer ( ObjA.meth100 ) ;
 }
}
```

Lors de l'exécution, nous avons vu qu'il y a création d'un ObjA de ClasseA et création d'un objet délégué FoncDeleg, les propriétés **Method** et **Target** sont automatiquement initialisées par le compilateur :



En fait, ce sont les champs privés qui sont initialisés et les propriétés Method et Target qui sont en lecture seulement, lisent les contenus respectifs de **_methodPtr** et de **_target;** le champ **_prev** est pour l'instant mis à **null** , enfin la méthode meth100(...) est actuellement en tête de liste.

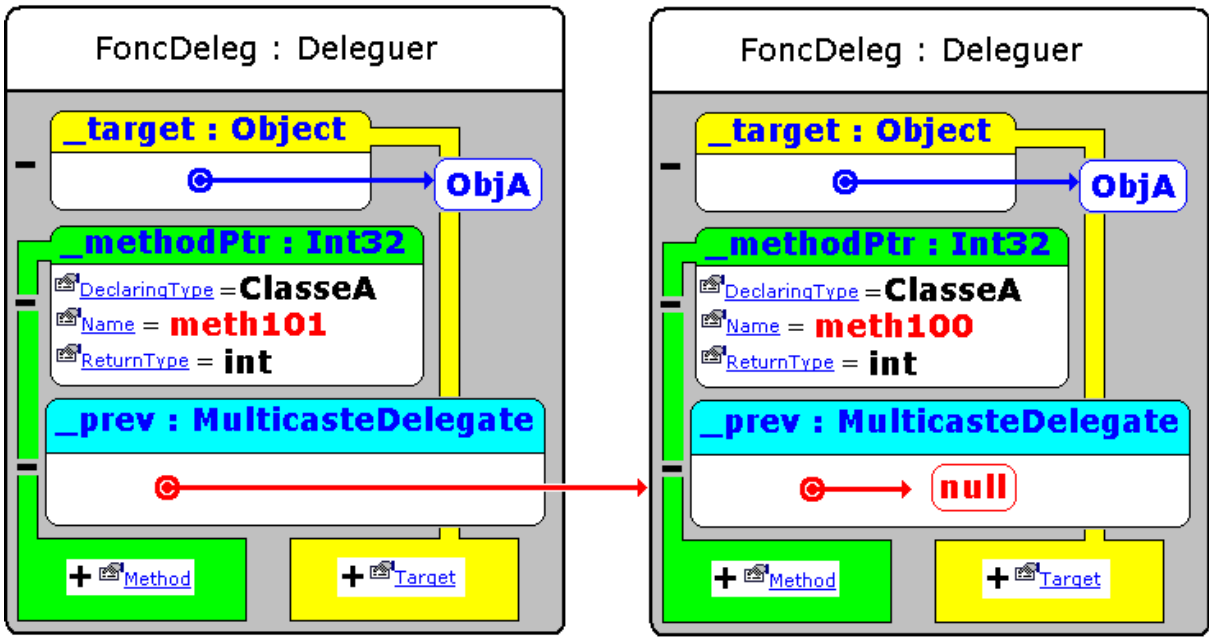
Figure virtuelle de l'objet délégué à ce stade :



Il est possible d'ajouter une nouvelle méthode meth101(...) au délégué qui va la mettre en tête de liste à la place de la méthode meth100(...) qui devient le deuxième élément de la liste. C# utilise l'opérateur d'addition pour implémenter l'ajout d'une nouvelle méthode au délégué. Nous étendons le programme précédent :

```
delegate int Deleguer ( char x );

class ClasseA {
 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 ;
 }
 static void Main ( string [] args ) {
   ClasseA ObjA = new ClasseA( );
   //-- meth100 est en tête de liste :
   Deleguer FoncDeleg = new Deleguer ( ObjA.meth100 ) ;
   // meth101 est ajoutée en tête de liste devant meth100 :
   FoncDeleg = FoncDeleg + new Deleguer ( ObjA.meth101 ) ;
 }
}
```

Figure virtuelle de l'objet délégué à cet autre stade :



C# permet de consulter et d'utiliser si nous le souhaitons toutes les référencement de méthodes en nous renvoyant la liste dans une tableau de référence de type **Delegate** grâce à la méthode **GetInvocationList**. Le source ci-dessous retourne dans le tableau Liste, la liste d'appel dans l'ordre d'appel, du délégué FoncDeleg :

```
Delegate[ ] Liste = FoncDeleg.GetInvocationList( ) ;
```

1.6 Exécution des méthodes d'un délégué multicast - Exemple de code

Lorsque l'on invoque le délégué sur un paramètre effectif, C# appelle et exécute séquentiellement les méthodes contenues dans la liste jusqu'à épuisement. **L'ordre d'appel est celui du stockage** : la première stockée est exécutée en premier, la suivante après, **la dernière méthode ajoutée est exécutée en dernier**, s'il y a un **résultat de retour, c'est celui de la dernière méthode ajoutée qui est renvoyé** , les autres résultats de retour sont ignorés. L'exemple ci-dessous reprend les notions que nous venons d'exposer.

Source complet exécutable d'un exemple de délégué multicast :

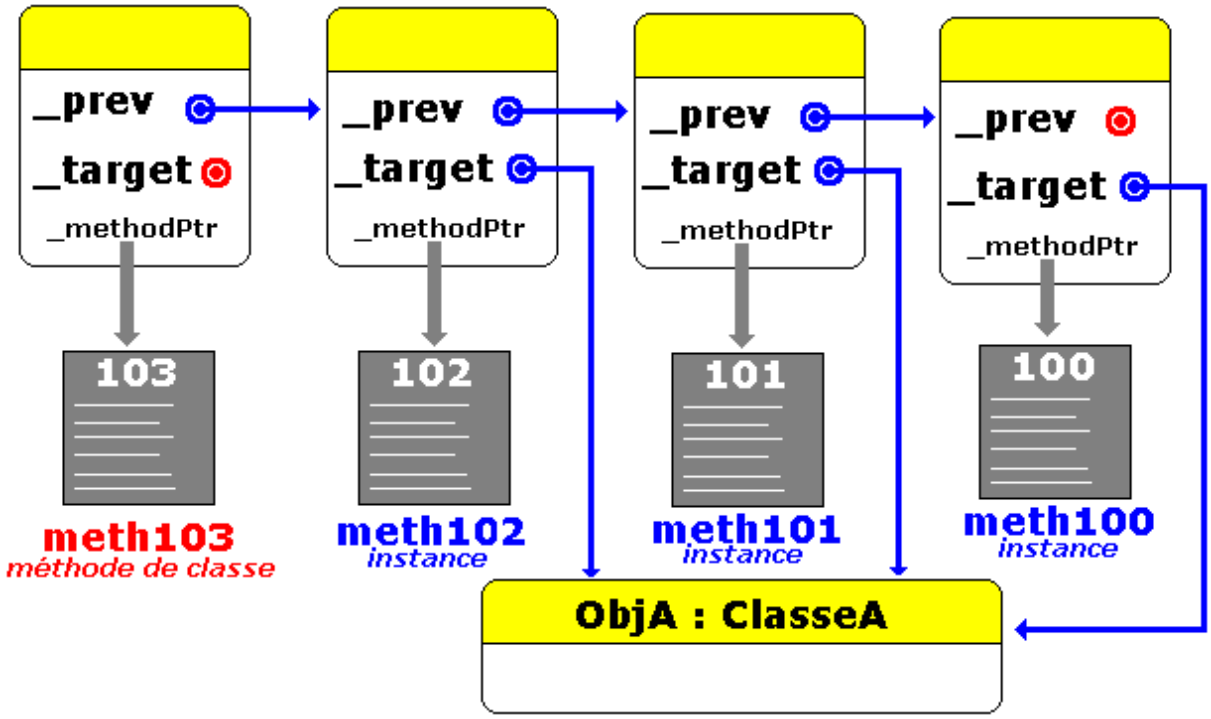
```
namespace PrDelegate {

 delegate  int  Deleguer ( 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 ;
  }

 static void  Main ( string [] args )  {
   System.Console.WriteLine ("Un délégué référence ObjA.meth1 :" ) ;
   ClasseA ObjX , ObjA = new ClasseA( );
   //-- instanciation du délégué avec ajout de 4 méthodes :
   Deleguer  FoncDeleg  =  new  Deleguer  (  ObjA.meth100  ) ;
   FoncDeleg +=  new  Deleguer  (  ObjA.meth101  ) ;
   FoncDeleg +=  new  Deleguer  (  ObjA.meth102  ) ;
   FoncDeleg +=  new  Deleguer  (  meth103  ) ;

   //--la méthode meth103 est en tête de liste :
   ObjX = (ClasseA)FoncDeleg.Target ;
   if (ObjX == null) System.Console.WriteLine ("Méthode static, Target = null") ;
   else if (ObjX.Equals(ObjA))System .Console.WriteLine ("Target référence bien ObjA") ;
   else System.Console.WriteLine ("Target ne référence pas ObjA") ;
   System.Console.WriteLine ( "\nnom : "+FoncDeleg.Method.Name );
   System.Console.WriteLine ( "classe : "+FoncDeleg.Method.DeclaringType.ToString( ) ) ;
   System.Console.WriteLine ( "retour : "+FoncDeleg.Method.ReturnType.ToString( ) );

   //--Appel du délégué sur le paramètre effectif 'a' :
   ObjA.champ = FoncDeleg('a') ; //code ascii 'a' = 97
   System.Console.WriteLine ( "\nvaleur du champ : "+ObjA.champ) ;
   System.Console.WriteLine ( "----------------------------------") ;

   //-- Parcours manuel de la liste des méthodes référencées :
   Delegate[ ] Liste = FoncDeleg.GetInvocationList( ) ;
   foreach ( Delegate Elt in Liste )
   {
    ObjX = (ClasseA)Elt.Target ;
    if (ObjX == null) System.Console.WriteLine ("Méthode static, Target = null") ;
    else if (ObjX.Equals(ObjA))System .Console.WriteLine ("Target référence bien ObjA") ;
    else System.Console.WriteLine ("Target ne référence pas ObjA") ;
    System.Console.WriteLine ( "\nnom : "+Elt.Method.Name ) ;
    System.Console.WriteLine ( "classe : "+Elt.Method.DeclaringType.ToString( ) ) ;
    System.Console.WriteLine ( "retour : "+Elt.Method.ReturnType.ToString( ) ) ;
    System.Console.WriteLine ( "----------------------------------") ;
   }
   System.Console.ReadLine ( ) ;
  }
 }
}
```

Résultats d'exécution sur la console :

```
D:\CsBuilder\Delegates\bin\Debug\PrDelegate.exe
Un délégué référence ObjA.meth1 :
Méthode static, Target = null

nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
Exécution de meth100('a')
Exécution de meth101('a')
Exécution de meth102('a')
Exécution de meth103('a')

valeur du champ : 200
----------------------------------
Target référence bien ObjA

nom : meth100
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
Target référence bien ObjA

nom : meth101
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
Target référence bien ObjA

nom : meth102
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
Méthode static, Target = null

nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
```

Nous voyons bien que le délégué **FoncDeleg** contient la liste des référencements des méthodes meth100, meth101,meth102 et meth103 ordonné comme figuré ci-dessous :



Remarquons que le premier objet de la liste est une référence sur une méthode de classe, la propriété Target renvoie la valeur **null** (le champ _target est à **null**).

La méthode de classe meth103 ajoutée en dernier est bien en tête de liste :

```
Un délégué référence ObjA.meth1 :
Méthode static, Target = null

nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
```

L'invocation du délégué lance l'exécution séquentielle des 4 méthodes :

```
Exécution de meth100('a')
Exécution de meth101('a')
Exécution de meth102('a')
Exécution de meth103('a')
```

et le retour du résutat est celui de meth103('a') :

```
valeur du champ : 200
----------------------
```

Le parcours manuel de la liste montre bien que ce sont des objets de type **Delegate** qui sont stockés et que l'on peut accéder entre autre possibilités, à leurs propriétés :

```
----------------------------------
Target référence bien ObjA

nom : meth100
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
Target référence bien ObjA

nom : meth101
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
Target référence bien ObjA

nom : meth102
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
Méthode static, Target = null

nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
----------------------------------
```

Les exceptions en

Plan général:

1. Les exceptions : syntaxe, rôle, classes

1.1 Comment gérer une exception dans un programme
1.2 Principe de fonctionnement de l'interception

2. Interception d'exceptions hiérarchisées

2.1 Interceptions de plusieurs exceptions
2.2 Ordre d'interception d'exceptions hiérarchisées

3. Redéclenchement d'une exception mot-clef : throw

3.1 Déclenchement manuel d'une exception de classe déjà existante
3.2 Déclenchement manuel d'une exception personnalisée

4. Clause finally

5. Un exemple de traitement d'exceptions sur des fichiers

6. Une solution de l'exemple précédent en C#

1. Les exceptions : syntaxe, rôle, classes

Rappelons au lecteur que la sécurité de fonctionnement d'une application peut être rendue instable par toute une série de facteurs :

Des problèmes liés au matériel : par exemple la perte subite d'une connexion à un port, un disque défectueux... Des actions imprévues de l'utilisateur, entraînant par exemple une division par zéro... Des débordements de stockage dans les structures de données...

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue, lors d'actions sur les données. Le logiciel doit donc se " ***défendre*** " contre de tels incidents potentiels, nous nommerons cette démarche la programmation défensive !

Programmation défensive
La ***programmation défensive*** est une attitude de pensée consistant à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

En programmation défensive il est possible de protéger directement le code à l'aide de la notion d'exception. L'objectif principal est d'améliorer la qualité de " ***robustesse*** " (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et efficace de logiciels robustes.

Rôle d'une exception
Une exception est chargée de signaler un comportement ***exceptionnel*** (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même. C'est le cas de C# qui intègre les exceptions comme une classe particulière: la classe **Exception**. Cette classe contient un nombre important de classes dérivées.

Comment agit une exception
Dès qu'une erreur se produit comme un manque de mémoire, un calcul impossible, un fichier inexistant, un transtypage non valide,..., un objet de la classe adéquate dérivée de la classe **Exception** est instancié. Nous dirons que le logiciel " ***déclenche une exception*** ".

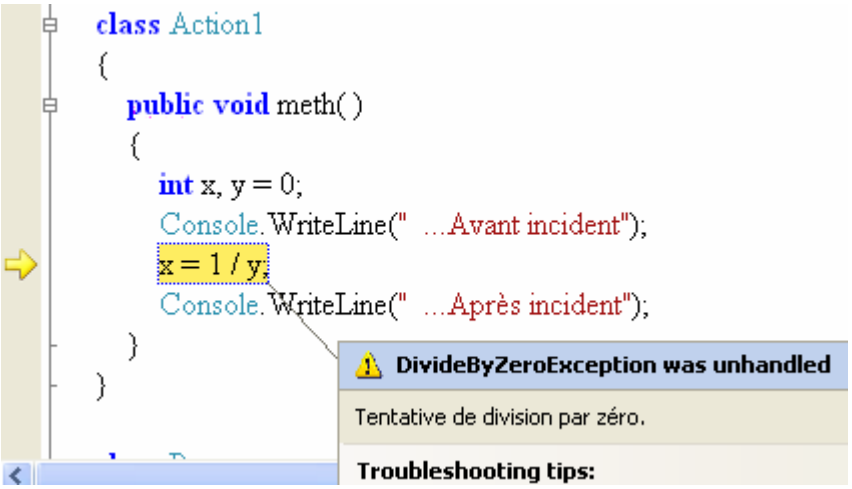
1.1 Comment gérer une exception dans un programme

Programme sans gestion de l'exception

Soit un programme C# contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;) dans la méthode meth() de la classe Action1, cette méthode est appelée dans la classe UseAction1 à travers un objet de classe Action1 :

```
class Action1 {
      public void meth()
      {
         int x, y =0;
         Console.WriteLine("  ...Avant incident");
         x = 1 / y;
         Console.WriteLine("  ...Après incident");
      }
 }

class Program {
      static void Main(string[] args)
      {
         Action1 Obj = new Action1( );
         Console.WriteLine("Début du programme.");
         Obj.meth( );
         Console.WriteLine("Fin du programme.");
      }
 }
```

Lors de l'exécution, après avoir affiché les chaînes "**Début du programme**" et " **...Avant incident**", le programme s'arrête et le CLR signale une erreur. Voici ci-dessous l'affichage obtenu dans Visual C# :



Sur la console l'affichage est le suivant :

> **Début du programme.**
> **...Avant incident.**
>
> *Le programme s'est arrêté à cet endroit et ne peut plus pouruivre son exécution.*

Que s'est-il passé ?

La méthode Main :
- a instancié un objet Obj de classe Action1,
- a affiché sur la console la phrase " Début du programme ",
- a invoqué la méthode meth() de l'objet Obj,
- a affiché sur la console la phrase " ...Avant incident",
- a exécuté l'instruction "x = 1/0;"

Dès que l'instruction "x = 1/0;" a été exécutée celle-ci a provoqué un incident. **En fait une exception de la classe DivideByZeroException a été "levée" (un objet de cette classe a été instancié) par le CLR**, cette classe hérite de la classe ArithmeticException selon la hiérarchie d'héritage suivante de .Net Framework :

```
System.Object
  |__System.Exception
      |__System.SystemException
          |__System.ArithmeticException
              |__System.DivideByZeroException
```

La classe mère de **toutes** les exceptions de .Net Framework est la classe **Exception**.

Le CLR a arrêté le programme immédiatement à cet endroit parce qu'elle n'a pas trouvé de code d'interception de cette exception qu'il a levée automatiquement :

```
class Action1 {
    public void meth()
    {
      int x, y =0;
      Console.WriteLine("  ...Avant incident");
      x = 1 / y;
      Console.WriteLine("  ...Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
      Action1 Obj = new Action1();
      Console.WriteLine("Début du programme.");
      Obj.meth();
      Console.WriteLine("Fin du programme.");
    }
}
```

Sortie du bloc

Sortie du bloc

Nous allons voir comment intercepter (on dit aussi "attraper" - to catch) cette exception afin de faire réagir notre programme pour qu'il ne s'arrête pas brutalement.

Programme avec gestion de l'exception

C# possède une instruction qui permet d'intercepter des exceptions dérivant de la classe Exception :

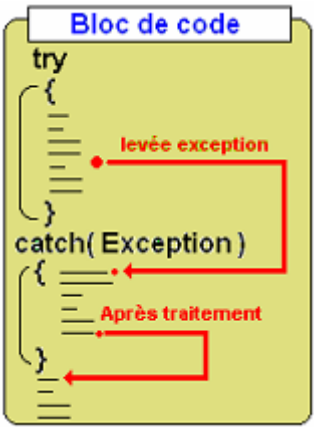
try ... catch

On dénomme cette instruction : un gestionnaire d'exceptions. Syntaxe **minimale** d'un tel gestionnaire try … catch :

```
try {
        <lignes de code à protéger>
    }catch ( UneException ) {
        <lignes de code réagissant à l'exception UneException >
    }
```

Le type **UneException** est obligatoirement une classe qui **hérite de la classe Exception**.

Schéma du fonctionnement d'un tel gestionnaire :



Le gestionnaire d'exception "déroute" l'exécution du programme vers le bloc d'interception catch qui traite l'exception (exécute le code contenu dans le bloc catch), puis renvoie et continue l'exécution du programme vers le code situé après le gestionnaire lui-même.

1.2 Principe de fonctionnement de l'interception

Dès qu'une **exception est levée** (instanciée), le CLR **stoppe immédiatement** l'exécution normale du programme à la **recherche d'un gestionnaire** d'exception susceptible d'intercepter (saisir) et de traiter cette exception. Cette recherche s'effectue à partir du **bloc englobant** et se poursuit sur les blocs plus englobants si aucun gestionnaire de **cette exception** n'a été trouvé.

Soit le même programme C# que précédemment, contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;). Cette fois nous allons gérer l'incident grâce à un gestionnaire d'exception **try..catch** dans le bloc englobant immédiatement supérieur**.**

Programme avec traitement de l'incident par try...catch :

```
class Action1 {
    public void meth()
    {
        int x, y =0;
        Console.WriteLine("  ...Avant incident");
        x = 1 / y;
        Console.WriteLine("  ...Après incident");
```

```
    }
}

class Program {
    static void Main(string[] args)
    {
        Action1 Obj = new Action1();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Interception exception");
        }
        Console.WriteLine("Fin du programme.");
    }
}
```

Figuration du déroulement de l'exécution de ce programme :

```
class Action1 {
    public void meth()
    {
        int x, y =0;
        Console.WriteLine("  ...Avant incident");
        x = 1 / y;          <-- engendre une exception
        Console.WriteLine("  ...Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
        Action1 Obj = new Action1();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();     --> levée d'une DivideByZeroException
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Interception exception");   --> traitement puis
        }
        Console.WriteLine("Fin du programme.");   <-- poursuite de l'exécution
    }
}
```

Ci-dessous l'affichage obtenu sur la console lors de l'exécution de ce programme :

Début du programme.
...Avant incident
Interception exception
Fin du programme.

- Nous remarquons que le CLR a donc bien exécuté le code d'interception situé dans le corps du "**catch** (DivideByZeroException){...}", il a poursuivi l'exécution normale après le gestionnaire.

- Le gestionnaire d'exception se situe dans la méthode Main (code englobant) qui appelle la méthode meth( ) qui lève l'exception.

Il est aussi possible d'atteindre l'objet d'exception qui a été instancié (ou levé) en déclarant un identificateur local au bloc catch du gestionnaire d'exception try … catch, cet objet est disponible dans tout le corps de la clause catch. Dans l'exemple qui suit, on déclare un objet Except de classe DivideByZeroException et l'on lit le contenu de deux de ses propriétés Message et Source :

```
try{
    Obj.meth();
}
catch ( DivideByZeroException Except ) {
   // accès aux membres publiques de l'objet Except :
   Console.WriteLine("Interception exception message : " + Except.Message);
   Console.WriteLine("Interception exception source : " + Except.Source);
   ….
}
```

2. Interception d'exceptions hiérarchisées

2.1 Interceptions de plusieurs exceptions

Dans un gestionnaire try...catch, il est en fait possible d'intercepter plusieurs types d'exceptions différentes et de les traiter.

Ci-après nous montrons la syntaxe d'un tel gestionnaire qui fonctionne comme un sélecteur ordonné, ce qui signifie qu'**une seule clause d'interception est exécutée**.

Dès qu'une exception intervient dans le < bloc de code à protéger>, le CLR scrute séquentiellement toutes les clauses **catch** de la première jusqu'à la nième. Si l'exception actuellement levée est d'un des types présents dans la liste des clauses le traitement associé est effectué, la scrutation est abandonnée et le programme poursuit son exécution après le gestionnaire.

```
try {
 < bloc de code à protéger>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
  .....
 catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

Où TypeException1, TypeException12, ... , TypeExceptionk sont des classes d'exceptions obligatoirement toutes **distinctes.**

Seule une seule clause **catch** (TypeException E) {...}est exécutée (celle qui correspond au bon type de l'objet d'exception instancié).

Exemple théorique :

Soit SystemException la classe des exceptions prédéfinies dans le nom d'espace System de .Net Framework, InvalidCastException, IndexOutOfRangeException, NullReferenceException et ArithmeticException sont des classes dérivant directement de la classe SystemException.

Supposons que la méthode meth() de la classe Action2 puisse lever quatre types différents d'exceptions: InvalidCastException, IndexOutOfRangeException, NullReferenceException et ArithmeticException.

Notre gestionnaire d'exceptions est programmé pour intercepter l'une de ces 4 catégories. Nous figurons ci-dessous les trois schémas d'exécution correspondant chacun à la levée (l'instanciation d'un objet) d'une exception de l'un des trois types et son interception :

```
class Action2    {
      public void meth( )
      {
         // une exception est levée .....
      }
   }

   class Program    {
      static void Main(string[] args)
      {
         Action2 Obj = new Action2();
         Console.WriteLine("Début du programme.");
         try
         {
            Obj.meth();
         }
         catch ( InvalidCastException E )
         {
            Console.WriteLine("Interception InvalidCastException");
         }
         catch ( IndexOutOfRangeException E )
         {
            Console.WriteLine("Interception IndexOutOfRangeException ");
         }
         catch ( NullReferenceException E )
         {
            Console.WriteLine("Interception NullReferenceException");
         }
         catch ( ArithmeticException E )
         {
            Console.WriteLine("Interception ArithmeticException");
         }
         Console.WriteLine("Fin du programme.");
      }
    }
```

Nous figurons ci-après deux schémas d'interception sur l'ensemble des quatre possibles d'une éventuelle exception qui serait levée dans la méthode meth( ) de l'objet Obj.

- **Schéma d'interception d'une IndexOutOfRangeException :**

```
static void Main(string[] args)
{
   Action2 Obj = new Action2();
   Console.WriteLine("Début du programme.");
   try
   {
      Obj.meth();
   }
   catch ( InvalidCastException E )
   {
      Console.WriteLine("Interception InvalidCastException");
   }
   catch ( IndexOutOfRangeException E )
   {
      Console.WriteLine("Interception IndexOutOfRangeException ");
   }
   catch ( NullReferenceException E )
   {
      Console.WriteLine("Interception NullReferenceException");
   }
   catch ( ArithmeticException E )
   {
      Console.WriteLine("Interception ArithmeticException");
   }
   Console.WriteLine("Fin du programme.");
}
```

levée d'une IndexOutOfRangeException

Traitement

poursuite de l'exécution après le bloc du gestionnaire

- **Schéma d'interception d'une ArithmeticException :**

```
static void Main(string[] args)
{
   Action2 Obj = new Action2();
   Console.WriteLine("Début du programme.");
   try
   {
      Obj.meth();
   }
   catch ( InvalidCastException E )
   {
      Console.WriteLine("Interception InvalidCastException");
   }
   catch ( IndexOutOfRangeException E )
   {
      Console.WriteLine("Interception IndexOutOfRangeException ");
   }
   catch ( NullReferenceException E )
   {
      Console.WriteLine("Interception NullReferenceException");
   }
   catch ( ArithmeticException E )
   {
      Console.WriteLine("Interception ArithmeticException");
   }
   Console.WriteLine("Fin du programme.");
}
```

levée d'une ArithmeticException

Traitement

poursuite de l'exécution après le bloc du gestionnaire

2.2 Ordre d'interception d'exceptions hiérarchisées

Dans un gestionnaire **try...catch** comprenant plusieurs clauses, la recherche de la clause catch contenant le traitement de la classe d'exception appropriée, s'effectue séquentiellement dans l'ordre d'écriture des lignes de code.

Soit le pseudo-code C# suivant :

```
try {
  < bloc de code à protéger générant un objet exception>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
  .....
catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

La recherche va s'effectuer comme si le programme contenait des **if...else if...** imbriqués :

```
if (<Objet exception> is TypeException1)  { <Traitement TypeException1 > }
else if (<Objet exception> is TypeException2)  { <Traitement TypeException2 > }
...
else if (<Objet exception> is TypeExceptionk)  { <Traitement TypeExceptionk > }
```

Les tests sont effectués sur l'appartenance de l'objet d'exception à une classe à l'aide de l'opérateur **is**.

> Signalons que l'opérateur **is** agit sur une classe et ses classes filles (sur une hiérarchie de classes), c'est à dire que tout objet de classe TypeExceptionX est aussi considéré comme un objet de classe parent au sens du test d'appartenance en particulier cet objet de classe TypeExceptionX est aussi considéré objet de classe Exception qui est la classe mère de toutes les exceptions C#.

Le test d'appartenance de classe dans la recherche d'une clause **catch** fonctionne d'une façon identique à l'opérateur **is** dans les if...else

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans l'ordre inverse de la hiérarchie.

Exemple : Soit une hiérarchie d'exceptions de C#

```
System.Exception
  |__System.SystemException
        |__System.InvalidCastException
        |__System.IndexOutOfRangeException
        |__System.NullReferenceException
        |__System.ArithmeticException
```

Soit le modèle de gestionnaire d'interception déjà fourni plus haut :

```
try {   < bloc de code à protéger générant un objet exception> }
catch (InvalidCastException E ) { <Traitement InvalidCastException > }
catch ( IndexOutOfRangeException E ) { <Traitement IndexOutOfRangeException > }
catch (NullReferenceException E ) { <Traitement NullReferenceException > }
```

catch (ArithmeticException E) { <Traitement ArithmeticException > }

Supposons que nous souhaitions intercepter une cinquième classe d'exception, par exemple une DivideByZeroException**,** nous devons rajouter une clause :

catch (DivideByZeroException E) { <Traitement DivideByZeroException > }

Insérons cette clause en dernier dans la liste des clauses d'interception :

```
try
  {
     Obj.meth();
  }
  catch ( InvalidCastException E )
  {
     Console.WriteLine("Interception InvalidCastException");
  }
  catch ( IndexOutOfRangeException E )
  {
     Console.WriteLine("Interception IndexOutOfRangeException ");
  }
  catch ( NullReferenceException E )
  {
     Console.WriteLine("Interception NullReferenceException");
  }
  catch ( ArithmeticException E )
  {
     Console.WriteLine("Interception ArithmeticException");
  }
 catch (DivideByZeroException E )
  {
     Console.WriteLine("Interception DivideByZeroException ");
  }
  Console.WriteLine("Fin du programme.");
}
```

Nous lançons ensuite la compilation de cette classe et nous obtenons un message d'erreur :

Error 1 : Une clause catch précédente intercepte déjà toutes les expressions de this ou d'un super type ('System.ArithmeticException')

Le compilateur proteste à partir de la clause **catch** (DivideByZeroException E) en nous indiquant que l'exception est déjà interceptée.

Que s'est-il passé ?

Le fait de placer en premier la clause **catch** (ArithmeticException E) chargée d'intercepter les exceptions de classe ArithmeticException implique que n'importe quelle exception héritant de ArithmeticException comme par exemple DivideByZeroException, est considérée comme une ArithmeticException. Dans un tel cas cette DivideByZeroException est interceptée par la clause **catch** (ArithmeticException E) mais elle **n**'est **jamais** interceptée par la clause **catch** (DivideByZeroException E) .

Le seul endroit où le compilateur C# acceptera l'écriture de la **catch** (DivideByZeroException E) se situe dans cet exemple **avant la clause catch** (ArithmeticException E). Ci-dessous l'écriture d'un programme correct :

```
try
  {
    Obj.meth();
  }
  catch ( InvalidCastException E )
  {
    Console.WriteLine("Interception InvalidCastException");
  }
  catch ( IndexOutOfRangeException E )
  {
    Console.WriteLine("Interception IndexOutOfRangeException ");
  }
  catch ( NullReferenceException E )
  {
    Console.WriteLine("Interception NullReferenceException");
  }
  catch (DivideByZeroException E )
  {
    Console.WriteLine("Interception DivideByZeroException ");
  }
  catch ( ArithmeticException E )
  {
    Console.WriteLine("Interception ArithmeticException");
  }
  Console.WriteLine("Fin du programme.");
}
```

Dans ce cas la recherche séquentielle dans les clauses permettra le filtrage correct des classes filles puis ensuite le filtrage des classes mères.

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs clauses dans l'ordre inverse de la hiérarchie.

3. Redéclenchement d'une exception mot-clef : throw

Il est possible de déclencher soi-même des exceptions en utilisant l'instruction **throw**, voir même de déclencher des exceptions personnalisées ou non. Une exception personnalisée est une classe héritant de la classe System.Exception définie par le développeur lui-même.

3.1 Déclenchement manuel d'une exception de classe déjà existante

Le CLR peut déclencher une exception automatiquement comme dans l'exemple de la levée d'une DivideByZeroException lors de l'exécution de l'instruction "x = 1/y ;".

Le CLR peut aussi lever (déclencher) une exception à votre demande suite à la rencontre d'une instruction **throw**. Le programme qui suit lance une ArithmeticException avec le message "Mauvais calcul !" dans la méthode **meth( )** et intercepte cette exception dans le bloc englobant **Main**. Le traitement de cette exception consiste à afficher le contenu du champ message de l'exception grâce à la propriété **Message** de l'exception :

```
class Action3
{
    public void meth()
    {
        int x = 0;
        Console.WriteLine("  ...Avant incident");
        if (x == 0)
            throw new ArithmeticException("Mauvais calcul !");
        Console.WriteLine("  ...Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
        Action3 Obj = new Action3();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();
        }
        catch (ArithmeticException E)
        {
            Console.WriteLine("Interception ArithmeticException : "+E.Message);
        }
        Console.WriteLine("Fin du programme.");
    }
}
```

Résultats de l'exécution du programme précédent :

Début du programme.
...Avant incident
Interception ArithmeticException : Mauvais calcul !
Fin du programme.

3.2 Déclenchement manuel d'une exception personnalisée

Pour une exception personnalisée, le mode d'action est strictement identique, il vous faut seulement auparavant créer une nouvelle classe **héritant obligatoirement de la classe Exception** ou de n'importe laquelle de ses sous-classes.

Reprenons le programme précédent et créons une classe d'exception que nous nommerons ArithmeticExceptionPerso héritant de la classe des ArithmeticException puis exécutons ce programme :

```
class ArithmeticExceptionPerso : ArithmeticException
{
    public ArithmeticExceptionPerso(String s) : base(s)
    {
    }
}

class Action3
{
    public void meth()
    {
        int x = 0;
        Console.WriteLine("  ...Avant incident");
```

```
        if (x == 0)
            throw new ArithmeticExceptionPerso ("Mauvais calcul !");
        Console.WriteLine("  ...Après incident");
    }
  }

class Program {

    static void Main(string[] args)
    {
        Action3 Obj = new Action3();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();
        }
        catch (ArithmeticExceptionPerso E)
        {
            Console.WriteLine("Interception ArithmeticExceptionPerso: "+E.Message);
        }
        Console.WriteLine("Fin du programme.");
    }
}
```

Résultats de l'exécution du programme précédent :

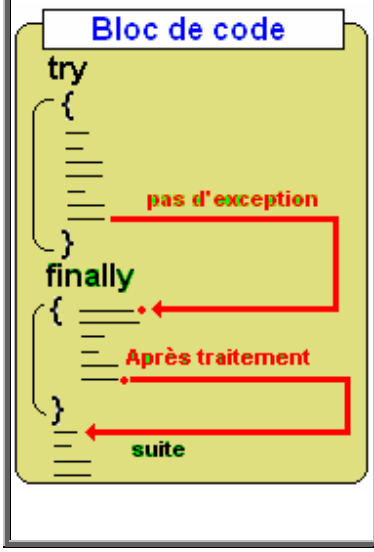
Début du programme.
...Avant incident
Interception ArithmeticExceptionPerso : Mauvais calcul !
Fin du programme.

L'exécution de ce programme est identique à celle du programme précédent, notre exception personnalisée fonctionne bien comme les exceptions prédéfinies de C#.

4. Clause finally

Supposons que nous soyons en présence d'un code contenant une éventuelle levée d'exception, mais supposons que quoiqu'il se passe nous désirions qu'un certain type d'action ait toujours lieu (comme par exemple fermer un fichier qui a été ouvert auparavant). Il existe en C# une **clause spécifique optionnelle** dans la syntaxe des gestionnaires d'exception permettant ce type de réaction du programme, c'est la clause **finally.** Voici en pseudo C#une syntaxe de cette clause :

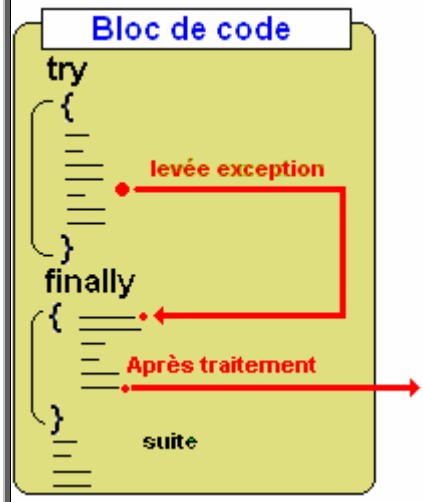
```
<Ouverture du fichier>
try {
    < action sur fichier>
}
finally {
   <fermeture du fichier>
}
.... suite
```



Fonctionnement sans incident

Si aucun incident ne se produit durant l'exécution du bloc < action sur fichier> :

l'exécution se poursuit à **l'intérieur du bloc finally par l'action** <fermeture du fichier> et **la suite** du code.



Fonctionnement si un incident se produit

Si un incident se produit durant l'exécution du bloc < action sur fichier> et qu'une exception est lancée:

malgré tout l'exécution se poursuivra à **l'intérieur du bloc finally par l'action** <fermeture du fichier>, puis **arrêtera** l'exécution du code dans le bloc.

La syntaxe C# autorise l'écriture d'une clause **finally** associée à plusieurs clauses **catch** :

```
try {
 <code à protéger>
}
catch (exception1 e ) { <traitement de l'exception1>  }
catch (exception2 e ) { <traitement de l'exception2>  }
...
finally {    <action toujours effectuée>  }
```

Remarque :

> Si le code du bloc à protéger dans try...finally contient une instruction de rupture de séquence comme break, return ou continue, le code de la clause finally{...} est malgré tout exécuté avant la rupture de séquence.

Nous avons vu lors des définitions des itérations **while**, **for** et de l'instruction **continue**, que l'équivalence suivante entre un **for** et un **while** valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue** (instruction forçant l'arrêt d'un tours de boucle et relançant l'itération suivante) :

*Equivalence incorrecte si Instr contient un **continue** :*

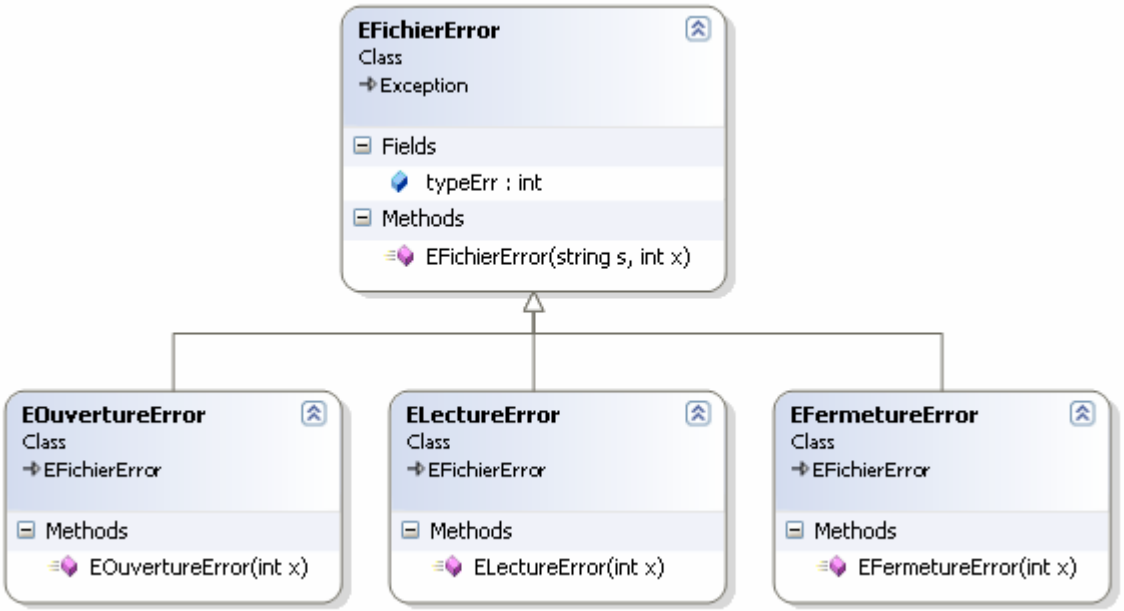
for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ; **while** (**Expr2**) { Instr ; **Expr3** }

*Equivalence correcte même si Instr contient un **continue** :*

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ;
	while (**Expr2**)
	{
	try {
	Instr ;
	} **finally** { **Expr3**
	}
	}

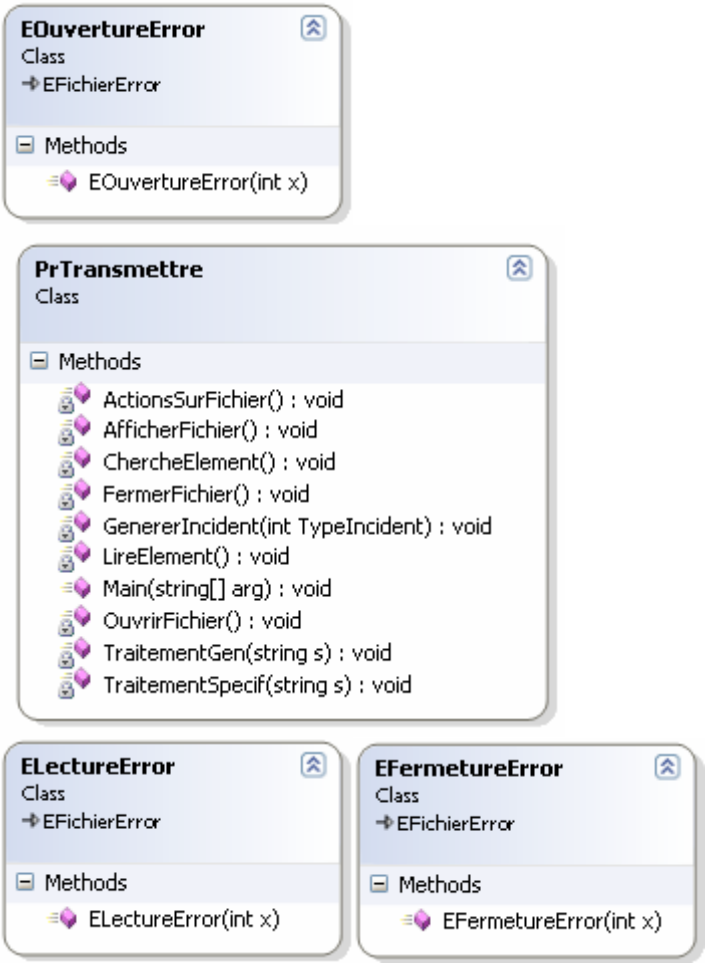
5. Un exemple de traitement d'exceptions sur des fichiers

Créons une hiérarchie d'exceptions permettant de signaler des incidents sur des manipulations de fichiers. Pour cela on distingue essentiellement trois catégories d'incidents qui sont représentés par trois classes :



Enoncé : Nous nous proposons de mettre en oeuvre les concepts précédents sur un exemple simulant un traitement de fichier. L'application est composée d'un bloc principal <programme> qui appelle une suite de blocs imbriqués.

Les classes en jeu et les blocs de programmes (méthodes) acteurs dans le traitement des exceptions sont figurées dans les diagrammes UML de droite :



<**programme: prTransmettre**>

…<**ActionsSurFichier**>

…..<**AfficheFichier**>

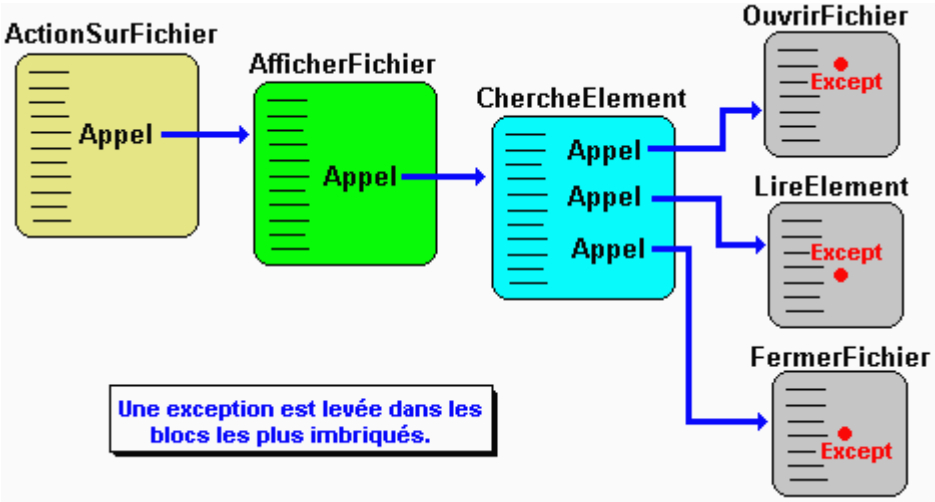
……...<**ChercheElement**>

.....<***OuvrirFichier***> **--> exception**

.....<***LireElement***> **--> exception**

….....<***FermerFichier***> **--> exception**

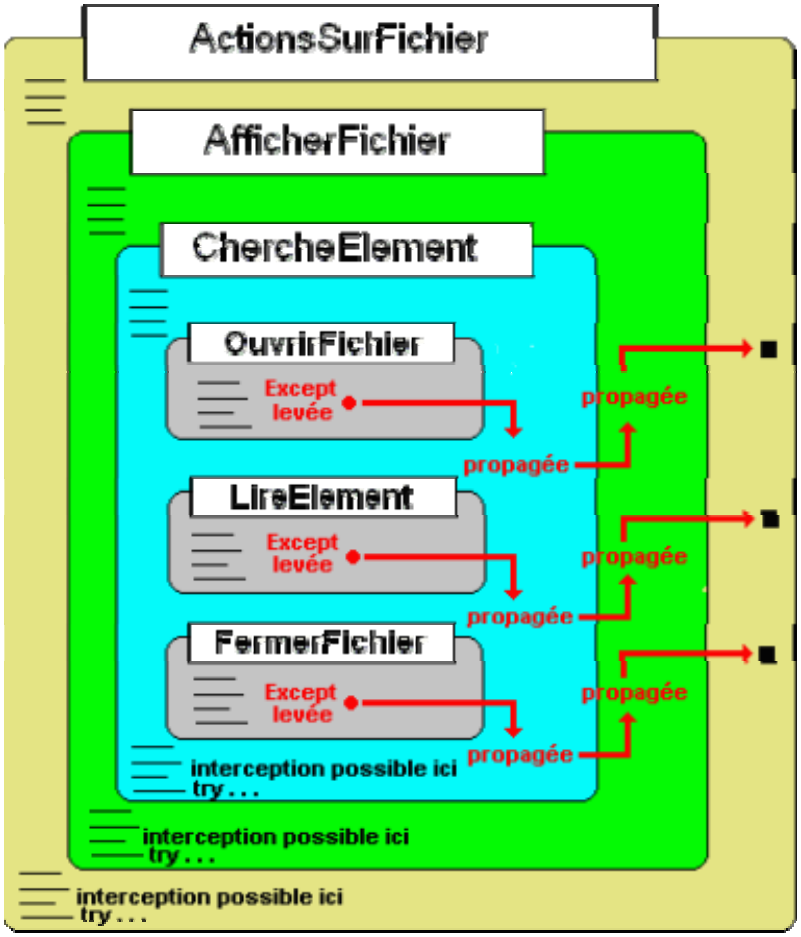
Les trois blocs du dernier niveau les plus internes <***OuvrirFichier***>, <***LireElement***> et <***FermerFichier***> peuvent lancer chacun une exception selon le schéma ci-après :



- ***La démarche***

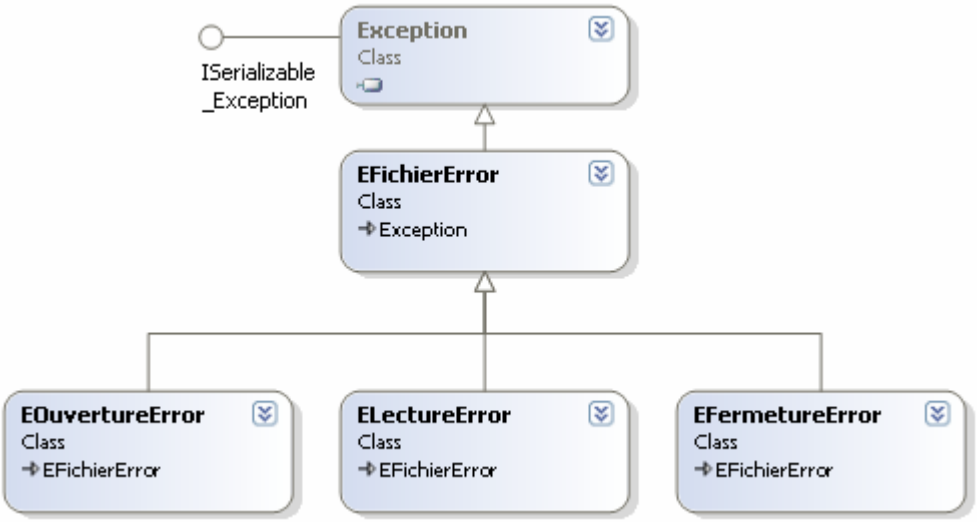
Les éventuelles exceptions lancées par les blocs <**OuvrirFichier**>,<**LireElement**> et <**FermerFichier**> doivent pouvoir se **propager** aux blocs de niveaux englobant afin d'être

interceptables à n'importe quel niveau.



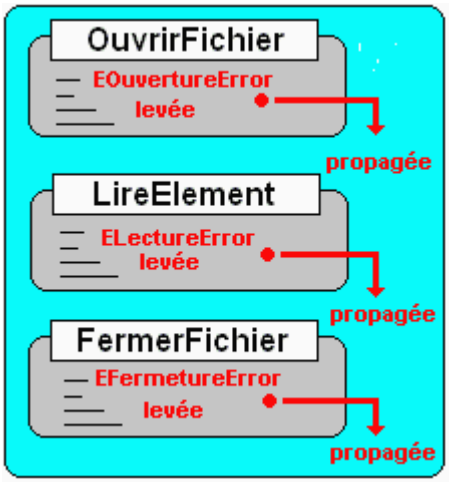
- ***Les classes d'exception***

On propose de créer une classe générale d'exception **EFichierError** héritant de la classe des **Exception**, puis 3 classes d'exception héritant de cette classe EFichierError :



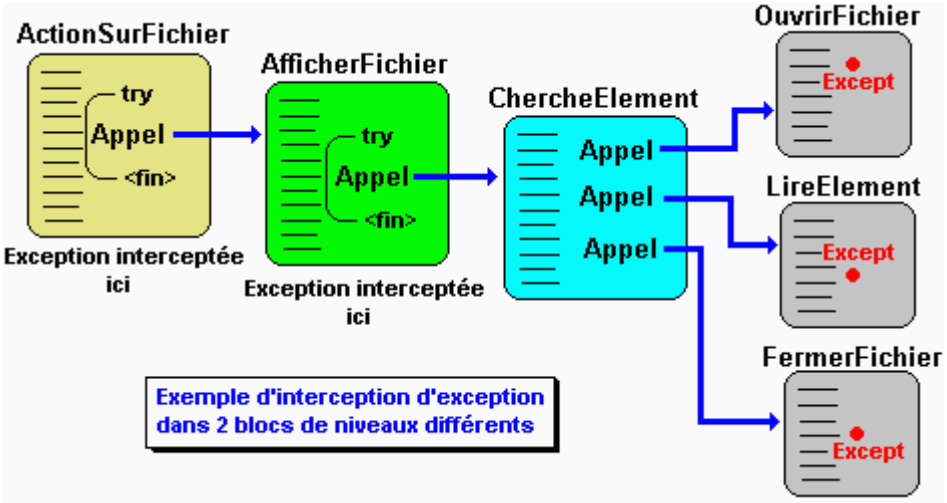
- ***Les blocs lançant éventuellement une exception***

Chaque bloc le plus interne peut lancer (lever) une exception de classe différente et la propage au niveau supérieur :



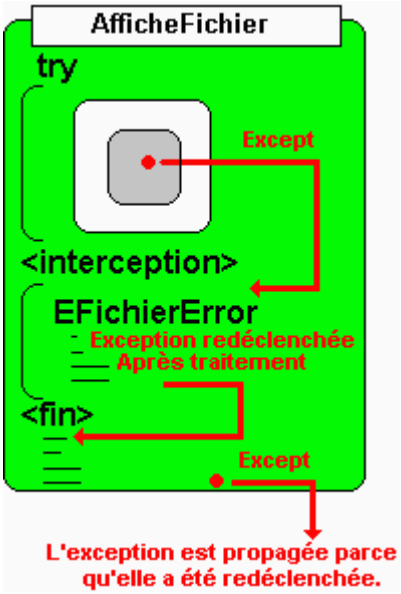
- ***Les blocs interceptant les exceptions***

Nous proposons par exemple d'intercepter les exceptions dans les deux blocs <**ActionsSurFichier**> et <**AfficheFichier**> :



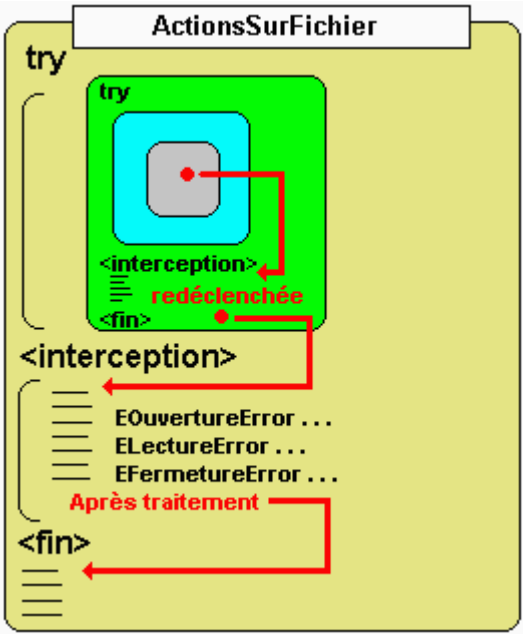
Le bloc <AfficherFichier>

Ce bloc interceptera une exception de type EFichierError, puis la redéclenchera après traitement :



Le bloc <ActionsSurFichier>

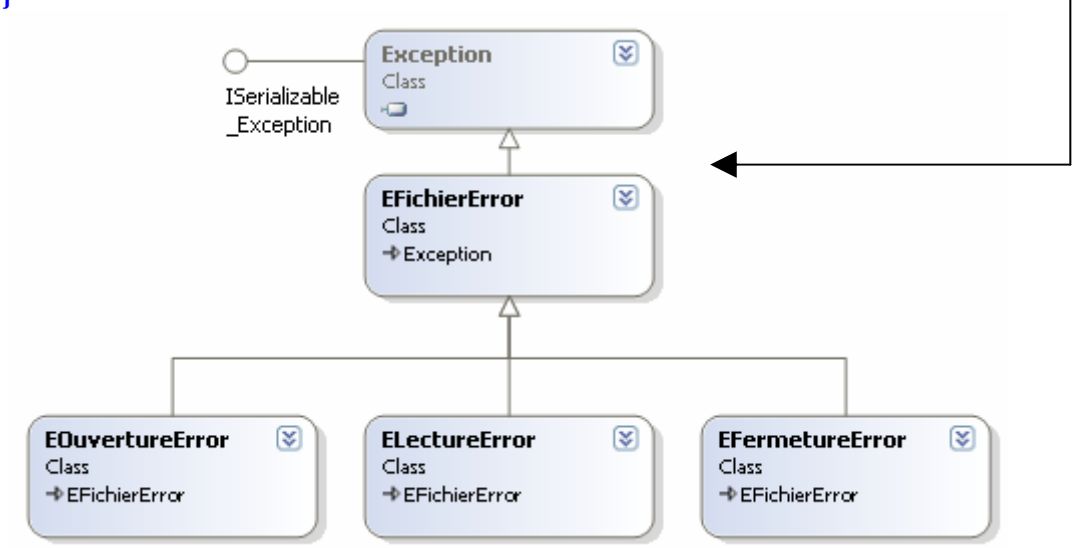
Ce bloc interceptera une exception de l'un des trois types EOuvertureError, ELectureError ou EFermetureError :



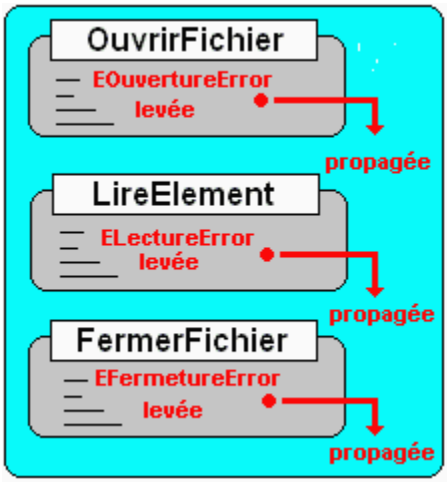
6. Une solution de l'exemple précédent en C#

```
using System ;
namespace PrTests
{
  /* pseudo-Traitement d'un fichier à plusieurs niveaux,
  * avec exception et relance d'exception
  */
 class EFichierError : Exception {
   public int typeErr ;
   public EFichierError ( String s, int x ): base ( s ) {
        typeErr = x ;
   }
}
```

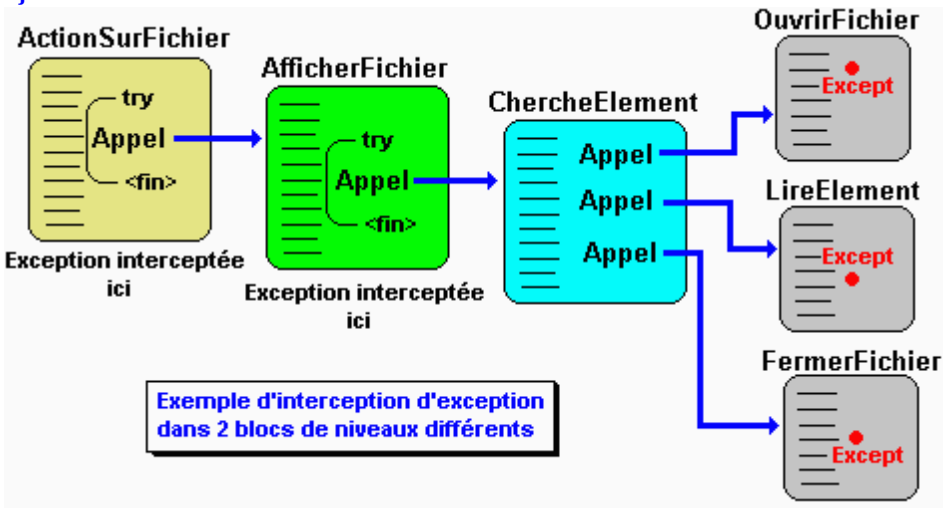
```
class EOuvertureError : EFichierError {
 public EOuvertureError ( int x ): base ("Impossible d'ouvrir le fichier !" ,x ) { }
}

class ELectureError : EFichierError{
 public ELectureError ( int x ): base ("Impossible de lire le fichier !" ,x ) { }
}

class EFermetureError : EFichierError{
 public EFermetureError ( int x ): base ("Impossible de fermer le fichier !" ,x ) {
 }
}
```




```
//-----------------------------------------------------------------------------------------------------------------------------------------------------------------

public class PrTransmettre {

 void TraitementGen ( String s ) {
  System .Console.WriteLine
("traitement general de l'erreur: " + s );
 }

 void TraitementSpecif ( String s ) {
  System .Console.WriteLine
("traitement specifique de l'erreur: " + s );
 }

 void OuvrirFichier ( ) {
  System .Console.WriteLine (" >> Action ouverture...");
  GenererIncident ( 1 );
  System .Console.WriteLine (" >> Fin ouverture.");
 }
void LireElement ( ) {
  System .Console.WriteLine (" >> Action lecture...");
  GenererIncident ( 2 );
  System .Console.WriteLine (" >> Fin lecture.");
 }

void FermerFichier ( ) {
  System .Console.WriteLine (" >> Action fermeture...");
  GenererIncident ( 3 );
  System .Console.WriteLine (" >> Fin fermeture.");
 }
```



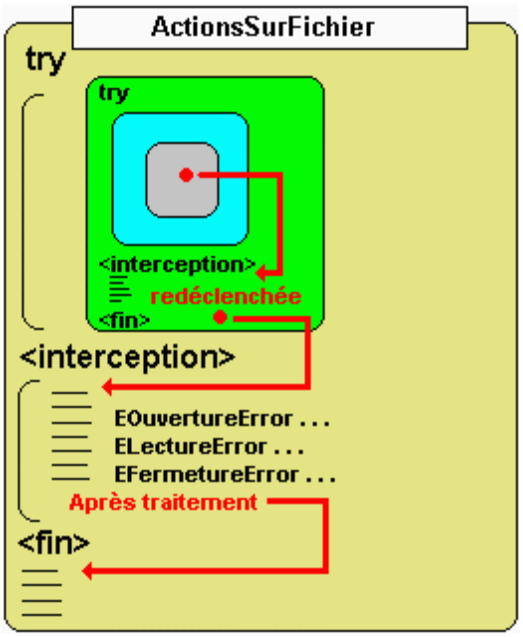
```
void  ChercheElement ( )  {
   OuvrirFichier ( );
   LireElement ( );
   FermerFichier ( );
 }
```



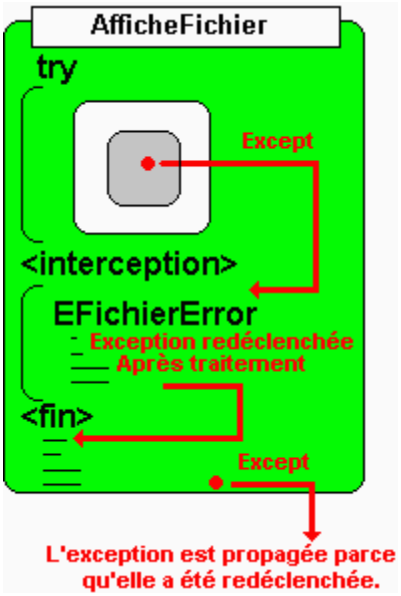

```
//-------------------------------------------------------------------------------------------

  void  GenererIncident ( int  TypeIncident )   {
   int  n ;
   Random nbr  =  new  Random ( );
  switch ( TypeIncident )
  {
   case  1 : n =  nbr.Next ( ) % 4 ;
    if ( n == 0 )
       throw new  EOuvertureError ( TypeIncident );
   break;
   case  2 : n =  nbr.Next ( ) % 3 ;
    if ( n == 0 )
       throw new  ELectureError ( TypeIncident );
   break;
   case  3 : n = nbr.Next ( ) % 2 ;
    if ( n == 0 )
       throw new  EFermetureError ( TypeIncident );
   break;
  }
 }
```

```
//-------------------------------------------------------------
```

```
void  ActionsSurFichier ( )  {
  System .Console.WriteLine ("Debut du travail sur le fichier.");

 try
    {
      System .Console.WriteLine ("........");
      AfficherFichier ( );
    }


      catch( EOuvertureError E )
      {
        TraitementSpecif ( E.Message );
      }

      catch( ELectureError E )
      {
        TraitementSpecif ( E.Message );
      }

      catch( EFermetureError E )
      {
        TraitementSpecif ( E.Message );
      }

    System.Console.WriteLine ("Fin du travail sur le fichier.");
}
```




```
//-------------------------------------------------------------------
void  AfficherFichier ( )   {

  try    {
    ChercheElement ( );
   }
   catch( EFichierError E )    {
    TraitementGen ( E.Message );
     throw  E ;
   }
  }

  public static void  Main ( string [ ] arg )  {
     PrTransmettre Obj  =  new  PrTransmettre ( );
    try    {
      Obj.ActionsSurFichier ( );
     }
     catch( EFichierError E )    {
      System .Console.WriteLine ( " Autre  type  d'Erreur
  générale Fichier !");
     }
      System .Console.ReadLine ( );
   }
  }
  }
```



Processus et multi-threading

Plan général:

1. Rappels

1.1 La multiprogrammation
1.2 Multitâche et processus
1.3 Multi-threading et processus

2. C# autorise l'utilisation des processus et des threads

2.1 Les processus avec C#
2.2 Comment exécuter une application à partir d'une autre application
2.3 Comment atteindre un processus déjà lancé
2.4 Comment arrêter un processus

3. C# et les threads

3.1 Comment créer un Thread
3.2 Comment endormir, arrêter ou interrompre un Thread
3.3 Exclusion mutuelle, concurrence, section critique, et synchronisation
3.4 Section critique en C# : lock
3.5 Section critique en C# : Monitor
3.6 Synchronisation commune aux processus et aux threads

1. Rappels

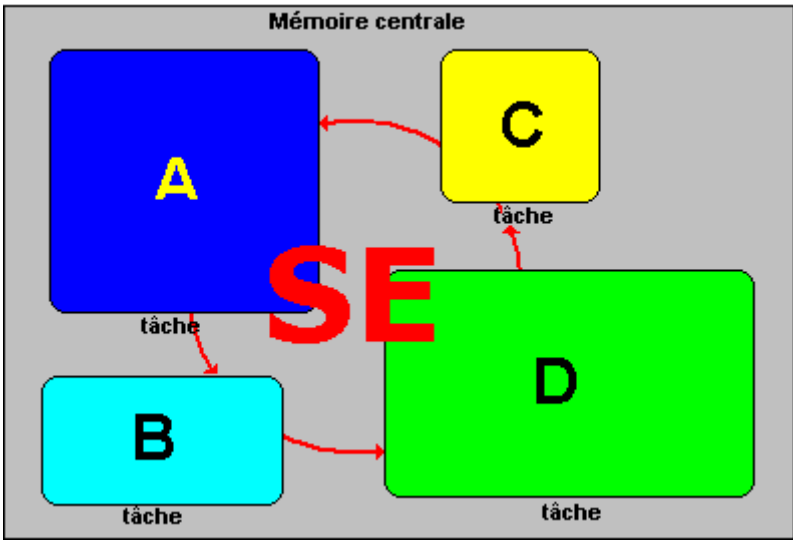
1.1 La multiprogrammation

Nous savons que les ordinateurs fondés sur les principes d'une machine de Von Neumann, sont des machines séquentielles donc n'exécutant qu'une seule tâche à la fois. Toutefois, le gaspillage de temps engendré par cette manière d'utiliser un ordinateur (le processeur central passe l'écrasante majorité de son temps à attendre) a très vite été endigué par l'invention de systèmes d'exploitations de multiprogrammation ou multitâches, permettant l'exécution "simultanée" de plusieurs tâches.

Dans un tel système, les différentes tâches sont exécutées sur une machine disposant d'**un seul processeur**, en apparence **en même temps** ou encore en **parallèle,** en réalité elles sont exécutées séquentiellement chacune à leur tour, ceci ayant lieu tellement vite pour notre conscience que nous avons l'impression que les programmes s'exécutent simultanément. Rappelons ici qu'une tâche est une application comme un traitement de texte, un navigateur Internet, un jeu,... ou d'autres programmes spécifiques au système d'exploitation que celui-ci exécute.

1.2 Multitâche et processus

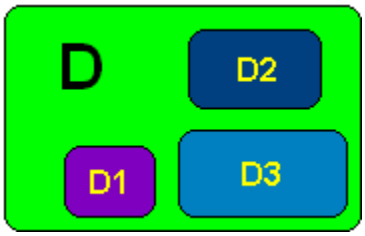
Le noyau du système d'exploitation SE, conserve en permanence le contrôle du temps d'exécution en distribuant cycliquement des tranches de temps (time-slicing) à chacune des applications A, B, C et D figurées ci-dessous. Dans cette éventualité, une application représente dans le système un **processus** :



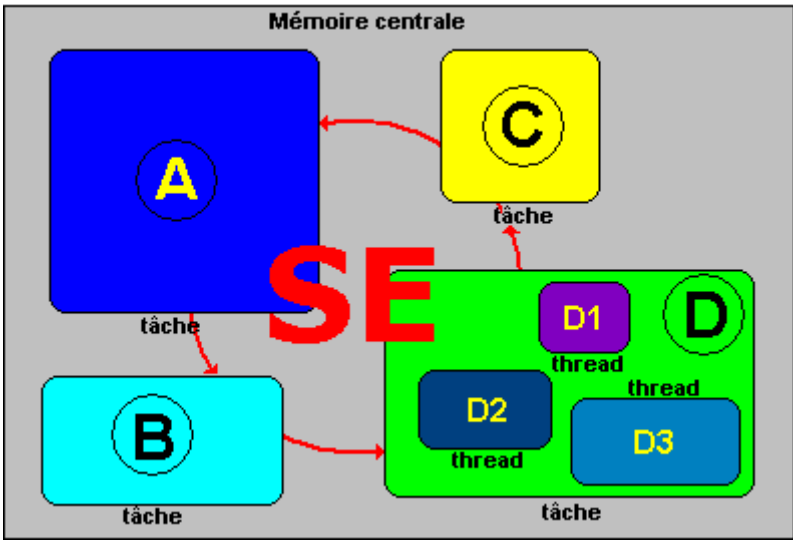
Rappelons la définition des **processus** donnée par A.Tanenbaum: un programme qui s'exécute et qui possède **son propre espace mémoire** : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multiprogrammation par la commutation entre processus effectuée par le processeur unique).

Thread
En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la multiprogrammation. Ces sous-tâches sont nommées "flux d'exécution" ou **Threads.**

Ci-dessous nous supposons que l'application D exécute en même temps les 3 Threads D1, D2 et D3 :

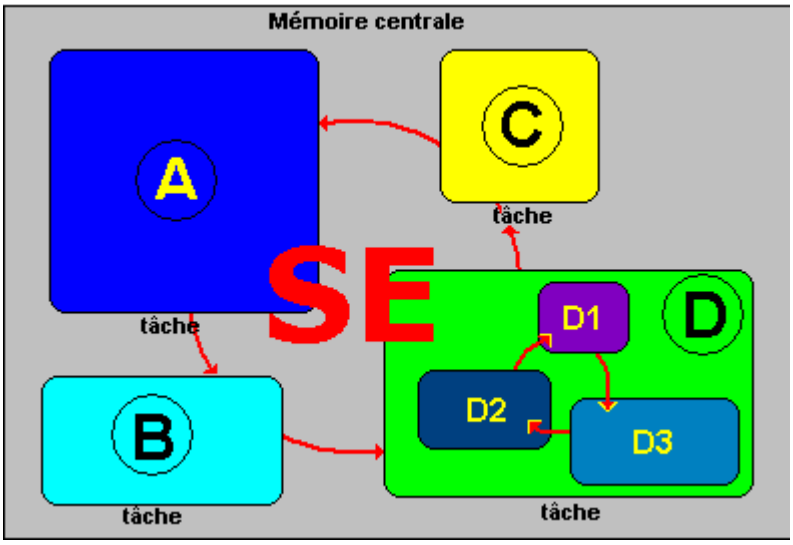


Reprenons l'exemple d'exécution précédent, dans lequel 4 processus s'exécutent "en même temps" et incluons notre processus D possédant 3 flux d'exécutions (threads) :



La commutation entre les threads d'un processus fonctionne de la même façon que la commutation entre les processus, chaque thread se voit alloué cycliquement, lorsque le processus D est exécuté une petite tranche de temps.

Le partage et la répartition du temps sont effectués **uniquement** par le système d'exploitation :



1.3 Multithreading et processus

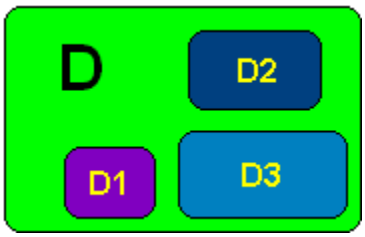
Définition :

La majorité des systèmes d'exploitation (Windows, Linux, Solaris, MacOs,...) supportent l'utilisation d'application contenant des threads, l'on désigne cette fonctionnalité sous le nom de **Multi-threading.**

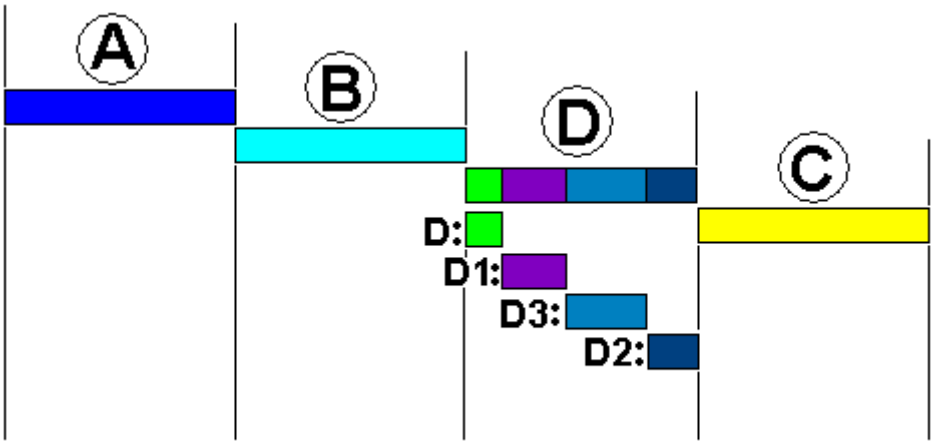
Différences entre threads et processus :

- Communication entre threads **plus rapide** que la communication entre processus,
- Les threads partagent un **même espace de mémoire** (de travail) entre eux,
- Les processus ont chacun un **espace mémoire personnel**.

Dans l'exemple précédent, figurons les processus A, B, C et le processus D avec ses threads dans un graphique représentant une tranche de temps d'exécution allouée par le système et supposée être la même pour chaque processus.

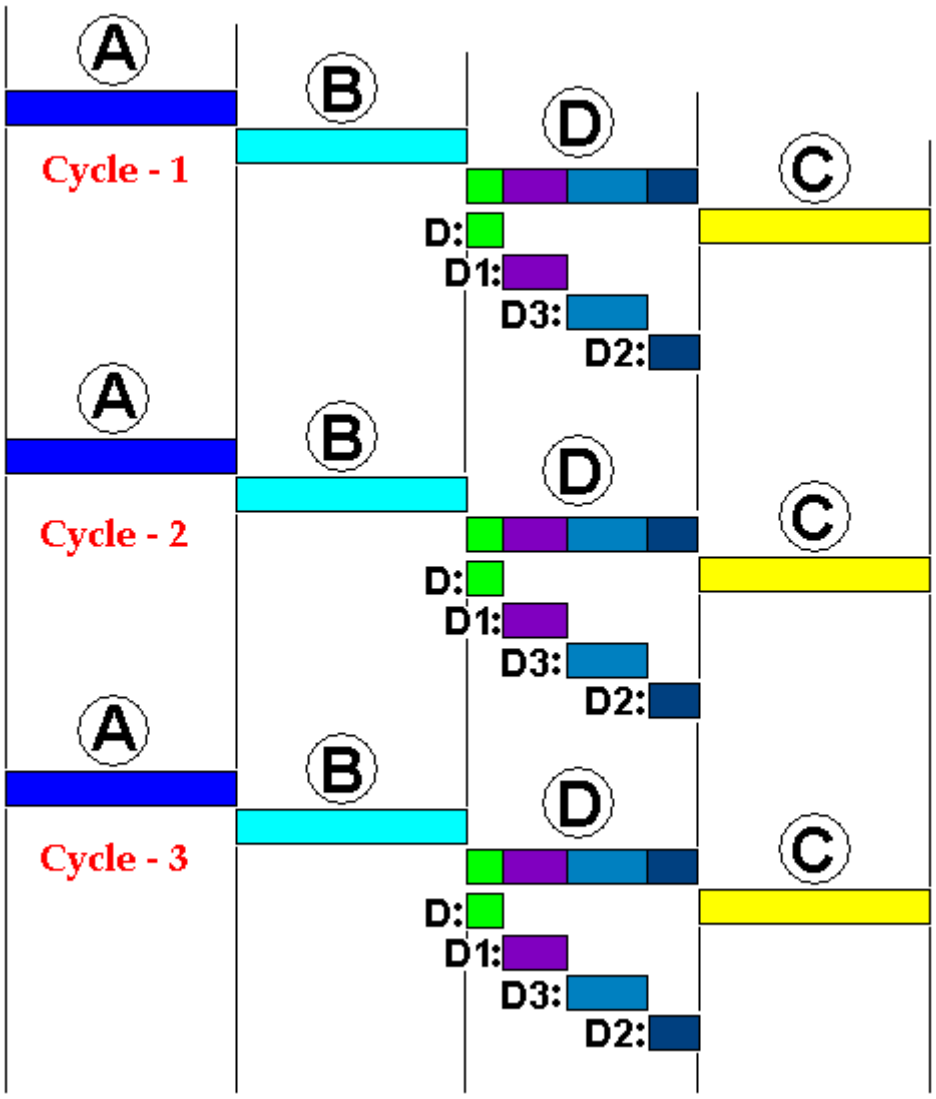


Le système ayant alloué le même temps d''exécution à chaque processus, lorsque par exemple le tour vient au processus D de s'exécuter dans sa tranche de temps, il exécutera une petite sous-tranche pour D1, pour D2, pour D3 et attendra le prochain cycle. Ci-dessous un cycle d'exécution :



Tranches de temps allouées pendant l'exécution

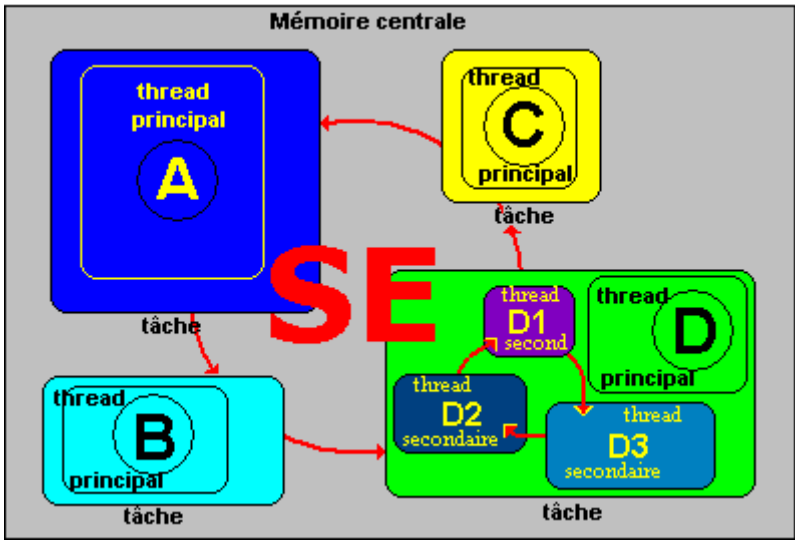
Voici sous les mêmes hypothèses de temps égal d'exécution alloué à chaque processus, le comportement de l'exécution sur 3 cycles consécutifs :



Le langages C# dispose de classes permettant d'écrire et d'utiliser des threads dans vos applications.

2. C# autorise l'utilisation des processus et des threads

Lorsqu'un programme C# s'exécute en dehors d'une programmation de multi-threading, le processus associé comporte automatiquement un thread appelé **thread principal**. Un autre thread utilisé dans une application s'appelle un thread secondaire. Supposons que les quatre applications (ou tâches) précédentes A, B, C et D soient toutes des applications C#, et que D soit celle qui comporte trois threads secondaires D1, D2 et D3 "parallèlement" exécutés :



2.1 Les processus avec C#

Il existe dans .Net Framework une classe nommée Process dans l'espace de nom System.Diagnostics, permettant d'accéder à des processus locaux à la machine ou distants et de les manipuler (démarrer, surveiller, stopper, …). Cette classe est bien documentée par la bibliothèque MSDN de Microsoft, nous allons dans ce paragraphe montrer comment l'utiliser à partir d'un cas pratique souvent rencontré par le débutant : comment lancer une autre application à partir d'une application déjà en cours d'exécution.

2.2 Comment exécuter une application à partir d'une autre

1°) Instancier un objet de classe Process :

```
Process AppliAexecuter = new Process( );
```

2°) Paramétrer les informations de lancement de l'application à exécuter :
Parmi les nombreuses propriétés de la classe Process la propriété StartInfo (**public** ProcessStartInfo StartInfo {**get**; **set**;}) est incontournable, car elle permet ce paramétrage.
Supposons que notre application se nomme "Autreappli.exe" qu'elle soit située sur le disque C: dans le dossier "Travail", qu'elle nécessite au démarrage comme paramètre le nom d'un fichier contenant des données, par exemple le fichier "donnees.txt" situé dans le dossier "C:\infos". La propriété StartInfo s'utilise alors comme suit afin de préparer le lancement de l'application :

```
AppliAexecuter.StartInfo.FileName ="c:\\Travail\\Autreappli.exe";
AppliAexecuter.StartInfo.UseShellExecute = false;
AppliAexecuter.StartInfo.RedirectStandardOutput = false;
Appliexec.StartInfo.Arguments ="c:\\infos\\donnees.txt";
```

Remarques :

- UseShellExecute = **false**: permet de lancer directement l'application sans avoir à utiliser l'interface shell du système d'exploitation.

- RedirectStandardOutput = **false**: la sortie standard du processus reste dirigée vers l'écran par défaut.

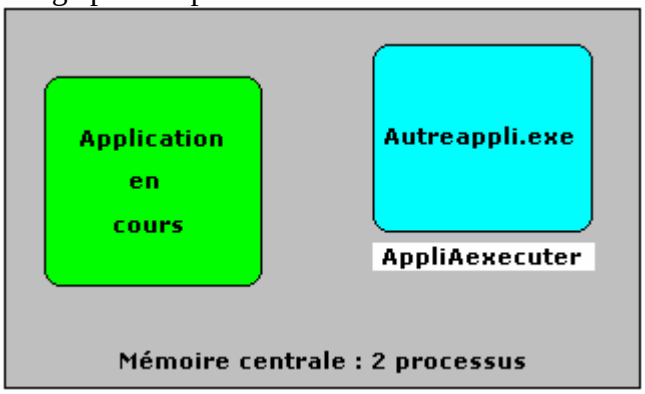
3°) Démarrer le processus par exemple par la surcharge d'instance de la méthode Start( ) de la classe Process :

```
AppliAexecuter.Start( );
```

Code récapitulatif de la méthode Main pour lancer un nouveau processus dans une application en cours d'exécution :

```
public static void Main( ) {
        Process AppliAexecuter = new Process( );
        AppliAexecuter.StartInfo.FileName ="c:\\Travail\\Autreappli.exe";
        AppliAexecuter.StartInfo.UseShellExecute = false;
        AppliAexecuter.StartInfo.RedirectStandardOutput = false;
        Appliexec.StartInfo.Arguments ="c:\\infos\\donnees.txt";
        AppliAexecuter.Start( );
}
```

Image produit par ce code dans la mémoire centrale :



2.3 Comment atteindre un processus déjà lancé

Atteindre le processus courant de l'application appelante :

```
Process Courant = Process.GetCurrentProcess();
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur l'ordinateur local :

```
Process [] localProcess = Process.GetProcessesByName("Autreappli");
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur un ordinateur distant de nom "ComputerDistant" :

```
Process [] localProcess = Process.GetProcessesByName("Autreappli","ComputerDistant");
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur un ordinateur distant d'adresse IP connue par exemple "101.22.34.18":

```
Process [] localProcess = Process.GetProcessesByName("Autreappli","101.22.34.18");
```

Atteindre un processus grâce à l'identificateur unique sur un ordinateur local de ce processus par exemple 9875 :

```
Process localProcess = Process.GetProcessById(9875);
```

Atteindre un processus grâce à l'identificateur unique de ce processus par exemple 9875 sur un ordinateur distant de nom "ComputerDistant" :

```
Process localProcess = Process.GetProcessById(9875, "ComputerDistant");
```

Atteindre un processus grâce à l'identificateur unique de ce processus par exemple 9875 sur un ordinateur distant d'adresse IP connue par exemple "101.22.34.18" :

```
Process localProcess = Process.GetProcessById(9875, "101.22.34.18");
```

2.4 Comment arrêter un processus

*Seuls les processus **locaux** peuvent être stoppés dans .Net Framework, les processus distants ne peuvent qu'être surveillés.*

Si le processus est une application fenêtrée (application possédant une interface IHM)

La méthode "**public bool** CloseMainWindow( )" est la méthode à employer

```
Process AppliAexecuter = new Process( );
….
AppliAexecuter.CloseMainWindow( );
```

Cette méthode renvoie un booléen qui indique :

- **True** si la fermeture a été correctement envoyée et le processus est stoppé.
- **False** si le processus n'est pas stoppé soit parcequ'il y a eu un incident, soit parce que le processus n'était une application fenêtrée (application console).

Si le processus n'est pas une application fenêtrée (application console)

La méthode "**public void** Kill( )" est la méthode à employer

```
Process AppliAexecuter = new Process( );
….
AppliAexecuter.Kill( );
```

Ou bien :

```
if( !AppliAexecuter.CloseMainWindow( ) )
        AppliAexecuter.Kill( );
```

3. C# et les threads

Comme l'avons déjà signalé tous les systèmes d'exploitation modernes permettent la programmation en multi-threading, le .Net Framework contient une classe réservée à cet usage dans l'espace de noms System.Threading : la classe non héritable Thread qui implémente l'interface _Thread.

```
public sealed class Thread: _Thread
```

Cette classe Thread sert à créer, contrôler, et modifier les priorités de threads.

3.1 Comment créer un Thread

Le C# 2.0 propose quatre surcharges du constructeur de Thread, toutes utilisent la notion de **delegate** pour préciser le code à exécuter dans le thread, nous examinons celle qui est la plus utilisée depuis la version 1.0.

```
public Thread ( ThreadStart start );
```

Le paramètre start de type ThreadStart est un objet **delegate** sans paramètre qui pointe sur (fait référence à) la méthode à appeler à chaque exécution du thread ainsi créé :

```
public delegate void ThreadStart ( );
```

Vous devez donc écrire une méthode de classe ou d'instance ayant la même signature que le delegate **void** ThreadStart ( ), puis créer l'objet **delegate** qui pointera vers cette méthode. Vous pouvez nommer cette méthode du nom que vous voulez, pour rester dans le style Java nous la dénommerons **run( )**. Ci-dessous un pseudo-code C# de création d'un thread à partir d'un délégué pointant sur une méthode de classe :

```
public class ChargerDonnees    {
  public static void run( ){  ....  }
}

public class AppliPrincipale    {
  public static void Main( ){
    ThreadStart ChargerDelegate = new ThreadStart (ChargerDonnees.run);
     Thread thrdChargement = new Thread(ChargerDelegate);
  }
}
```

Notons qu'il est possible d'alléger le code d'instanciation du thread en créant un objet délégué anonyme qui est passé en paramètre au constructeur de Thread :

```
Thread thrdChargement = new Thread( new ThreadStart (ChargerDonnees.run) );
```

Dans l'exemple qui suit nous lançons trois threads en plus du thread principal automatiquement construit par le CLR pour chaque processus, l'un à partir d'une méthode **static** run0 de la classe Program et les deux autres à partir d'une méthode **static** run1 et d'une méthode d'instance run2 de

la classe AfficherDonnees :

```
public class AfficherDonnees
   {
      public static void run1()
      {
         for (int i1 = 1; i1 < 100; i1++)
            System.Console.WriteLine(">>>  thread1 = " + i1);
      }
      public void run2()
      {
         for (int i2 = 1; i2 < 100; i2++)
            System.Console.WriteLine("***  thread2 = " + i2);
      }
   }
   public class Program
   {
      public static void run0()
      {
         for (int i0 = 1; i0 < 100; i0++)
            System.Console.WriteLine(".... thread0 = " + i0);
      }
      static void Main(string[] args)
      {
         Console.WriteLine("Lancement des threads :");
         AfficherDonnees obj = new AfficherDonnees();
         Thread thread0 = new Thread(new ThreadStart( run0 ));
         Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
         Thread thread2 = new Thread(new ThreadStart(obj.run2));
         thread0.Start();
         thread1.Start();
         thread2.Start();
         for (int i = 1; i < 100; i++)
            System.Console.WriteLine(" i = " + i);
         System.Console.WriteLine("fin de tous les threads.");
      }
   }
```

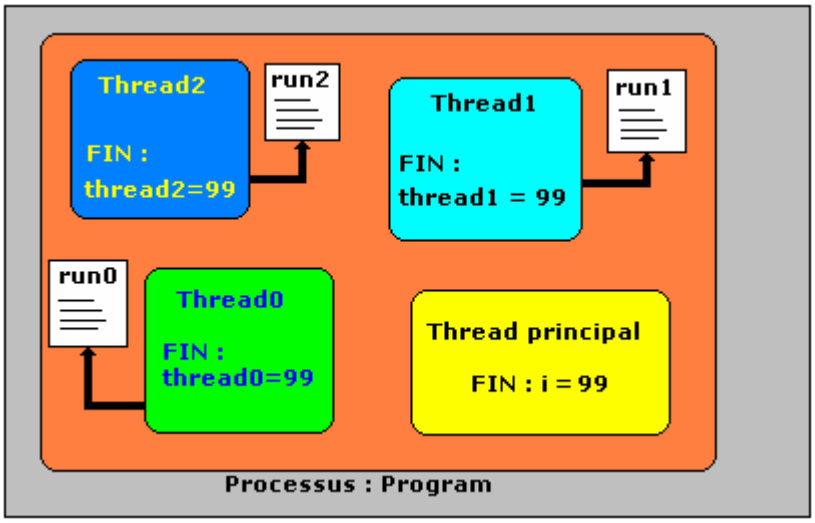
Résultats de l'exécution du programme précédent (dépendants de votre configuration machine+OS) :

```
Lancement des threads :
....    thread0 = 1
>>>  thread1 = 1
 i = 1
***   thread2 = 1
....    thread0 = 2
>>>  thread1 = 2
 i = 2
***  thread2 = 2
....   thread0 = 3
………————————————
i = 98
***   thread2 = 98
....    thread0 = 99
>>>  thread1 = 99
 i = 99
***   thread2 = 99
fin de tous les threads.
```

L'exécution précédente montre bien que chacun des thread0, thread1 et thread2 se voient allouer

une tranche de temps pendant laquelle chacun d'eux exécute une partie de sa boucle **for** et imprime ses informations sur la console. Chaque thread se termine lorsque le code de la méthode run vers laquelle il pointe a fini de s'exécuter.

Nous figurons ci-après l'image de la mémoire centrale pour le processus Program avec chacun des 3 threads instanciés pointant vers la méthode qu'il exécute :



3.2 Comment endormir, arrêter ou interrompre un Thread

La société Microsoft a déprécié depuis la version 2.0 du .Net Framework les méthodes Suspend() et Resume() qui permettaient d'effectuer la synchronisation entre les threads, toutefois elles n'ont pas été supprimées. En ce sens Microsoft adopte la même attitude que Sun pour Java, afin de ne pas encourager les développeurs à utiliser des outils qui se sont montrés sensibles aux blocages du type verrou mortel. Nous ne proposerons donc pas ces méthodes dangereuses au lecteur, mais plutôt des méthodes sûres.

Endormir : Méthode Sleep (…)

On parle d'endormir un thread pendant un certain temps **t**, lorsque l'exécution de ce thread est arrêtée pendant ce temps **t**, c'est à dire que le thread est retiré de la file d'attente de l'algorithme d'ordonnancement du système. A la fin du temps **t**, le thread est automatiquement "réveillé" par le système, c'est à dire qu'il est replacé dans la file d'attente de l'ordonnanceur et donc son exécution repart.

La méthode "**public static void** Sleep(**int** millisecondsTimeout)" sert à endormir un thread pendant un temps exprimé en millisecondes. Dans la méthode run0( ) de l'exemple précédent si nous rajoutons l'instruction "Thread.Sleep(2);", nous "ralentissons" l'exécution de la boucle, puisque à tous les tours de boucles nous bloquons le thread qui l'exécute pendant 2 ms.

```
public static void run0( )
{
   for (int i0 = 1; i0 < 100; i0++)
   {
      System.Console.WriteLine(".... thread0 = " + i0);
      Thread.Sleep(2);
```

```
        }
    }
```

Résultats de l'exécution du programme précédent avec Thread.Sleep(2) dans la méthode run0( ) ***:***

Lancement des threads : **.... thread0 = 1** **>>> thread1 = 1** **i = 1** ***** thread2 = 1** **……** **i = 98** ***** thread2 = 98** **.... thread0 = 79** **>>> thread1 = 99** **i = 99** ***** thread2 = 99** **.... thread0 = 80** **fin de tous les threads.** **.... thread0 = 81** **.... thread0 = 82** **.... thread0 = 83** **.... thread0 = 84**	**.... thread0 = 85** **.... thread0 = 86** **.... thread0 = 87** **.... thread0 = 88** **.... thread0 = 89** **.... thread0 = 90** **.... thread0 = 91** **.... thread0 = 92** **.... thread0 = 93** **.... thread0 = 94** **.... thread0 = 95** **.... thread0 = 96** **.... thread0 = 97** **.... thread0 = 98** **.... thread0 = 99** ***(résultats dépendants de votre configuration machine+OS)***

Notons que cette exécution est semblable à la précédente, du moins au départ, car nous constatons vers la fin que le thread0 a pris un léger retard sur ses collègues puisque le thread1 et le thread2 se termine avec la valeur 99, le thread principal affiche la phrase "**fin de tous les threads**" alors que le thread0 n'a pas encore dépassé la valeur 80.

Les trois threads thread1, thread2 et le thread principal sont en fait terminés seul le thread0 continue sont exécution jusqu'à la valeur 99 qui clôt son activité.

Arrêter : Méthode Abort (…)

Dans l'exemple précédent le message "**fin de tous les threads**" n'est pas conforme à la réalité puisque le programme est bien arrivé à la fin du thread principal, mais thread0 continue son exécution. Il est possible de demander au système d'arrêter définitivement l'exécution d'un thread, cette demande est introduite par la méthode d'instance **Abort** de la classe Thread qui lance le processus d'arrêt du thread qui l'appelle.

Nous reprenons le programme précédent dans lequel nous lançons une demande d'arrêt du thread0 par l'instruction : thread0.Abort();

```
public class AfficherDonnees
  {
    public static void run1()
    {
      for (int i1 = 1; i1 < 100; i1++)
        System.Console.WriteLine(">>>  thread1 = " + i1);
    }
    public void run2()
    {
      for (int i2 = 1; i2 < 100; i2++)
        System.Console.WriteLine("***  thread2 = " + i2);
    }
  }
```

```
public class Program
{
   public static void run0()
   {
      for (int i0 = 1; i0 < 100; i0++)
         System.Console.WriteLine(".... thread0 = " + i0);
         Thread.Sleep(2);
   }
   static void Main(string[] args)
   {
      Console.WriteLine("Lancement des threads :");
      AfficherDonnees obj = new AfficherDonnees( );
      Thread thread0 = new Thread(new ThreadStart( run0 ));
      Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
      Thread thread2 = new Thread(new ThreadStart(obj.run2));
      thread0.Start( );
      thread1.Start( );
      thread2.Start( );
      for (int i = 1; i < 100; i++)
         System.Console.WriteLine(" i = " + i);
      thread0.Abort();
      System.Console.WriteLine("fin de tous les threads.");
   }
}
```

Résultats de l'exécution du programme précédent avec Thread.Sleep(2) et thread0.Abort() ***:***

Lancement des threads :
.... thread0 = 1
>>> thread1 = 1
i = 1
***** thread2 = 1**
……
i = 98
***** thread2 = 98**
.... thread0 = 79
>>> thread1 = 99
i = 99
***** thread2 = 99**
.... thread0 = 80
fin de tous les threads. ***(Résultats dépendants de votre configuration machine+OS)***

Le thread0 a bien été arrêté avant sa fin normale, avec comme dernière valeur 80 pour l'indice de la boucle **for**.

Attendre : Méthode Join (…)

Un thread peut se trouver dans des états d'exécution différents selon qu'il est actuellement en cours d'exécution, qu'il attend, qu'il est arrêté etc… Il existe en C# un type énuméré ThreadState qui liste toutes les valeurs possibles des états d'un thread :

public enum ThreadState { Running = 0, StopRequested = 1, SuspendRequested = 2, Background = 4, Unstarted = 8, Stopped = 16, WaitSleepJoin = 32, Suspended = 64, AbortRequested = 128, Aborted = 256 }

Dans la classe Thread, nous trouvons une propriété "**public** ThreadState ThreadState {**get**;}" en lecture seule qui fournit pour un thread donné son état d'exécution. En consultant cette propriété le

développeur peut connaître l'état "en direct" du thread, notons que cet état peut varier au cours du temps.

On peut faire attendre la fin d'exécution complète d'un thread pour qu'un autre puisse continuer son exécution, cette attente est lancée par l'une des trois surcharges de la méthode d'instance **Join** de la classe Thread :

> **public void** Join();

Dans le programme précédent rappelons-nous que le thread0 prend du "retard" sur les autres car nous l'avons ralenti avec un Sleep(2). Au lieu de l'arrêter définitivement avant la dernière instruction de la méthode Main remplaçons l'instruction "thread0.Abort();"par l'instruction "thread0.Join();".

Que se passe-t-il : Le thread principal qui exécute la méthode main, invoque la méthode **Join** du thread0 avant de terminer son exécution, ce qui signifie que le thread principal bloque tant que le thread0 n'a pas fini complètement son exécution. Nous ajoutons au programme précédent une méthode **public static void** etatsThreads qui affiche l'état d'exécution du thread principal et des 3 threads instanciés :

```
public static void etatsThreads(Thread principal, Thread thrd1, Thread thrd2, Thread thrd3 )
{
        System.Console.WriteLine(principal.Name + " : " + principal.ThreadState);
        System.Console.WriteLine(thrd1.Name + " : " + thrd1.ThreadState);
        System.Console.WriteLine(thrd2.Name + " : " + thrd2.ThreadState);
        System.Console.WriteLine(thrd3.Name + " : " + thrd3.ThreadState);
}

static void Main(string[] args)
{
        Console.WriteLine("Lancement des threads :");
        AfficherDonnees obj = new AfficherDonnees();
        Thread principal = Thread.CurrentThread;
        principal.Name = "principal";
        Thread thread0 = new Thread(new ThreadStart(run0));
        thread0.Name = "thread0";
        Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
        thread1.Name = "thread1";
        Thread thread2 = new Thread(new ThreadStart(obj.run2));
        thread2.Name = "thread2";
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Start();
        thread1.Start();
        thread2.Start();
        etatsThreads(principal, thread0, thread1, thread2);
        for (int i = 1; i < 100; i++)
        {
           System.Console.WriteLine(" i = " + i);
        }
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Join();
        etatsThreads(principal, thread0, thread1, thread2);
        System.Console.WriteLine("fin de tous les threads.");
    }
```

Nous obtenons une référence sur le thread principal par la propriété CurrentThread dans

l'instruction : Thread principal = Thread.CurrentThread.

Résultats de l'exécution du programme précédent avec Thread.Sleep(2) et thread0.Join() :

Lancement des threads **principal : Running** **thread0 : Unstarted** **thread1 : Unstarted** **thread2 : Unstarted** **.... thread0 = 1** **>>> thread1 = 1** **principal : Running** ***** thread2 = 1** **>>> thread1 = 2** **.... thread0 = 2** **thread0 : WaitSleepJoin** ***** thread2 = 2** **>>> thread1 = 3** **.... thread0 = 3** **thread1 : WaitSleepJoin** ***** thread2 = 3** **>>> thread1 = 4** **thread2 : WaitSleepJoin** **.... thread0 = 4** ***** thread2 = 4** **>>> thread1 = 5** **i = 1** **……**	**i = 99** **principal : Running** **.... thread0 = 85** **thread0 : WaitSleepJoin** **thread1 : Stopped** **.... thread0 = 86** **thread2 : Stopped** **.... thread0 = 87** **.... thread0 = 88** **.... thread0 = 89** **.... thread0 = 90** **.... thread0 = 91** **.... thread0 = 92** **.... thread0 = 93** **.... thread0 = 94** **.... thread0 = 95** **.... thread0 = 96** **.... thread0 = 97** **.... thread0 = 98** **.... thread0 = 99** **principal : Running** **thread0 : Stopped** **thread1 : Stopped** **thread2 : Stopped** **fin de tous les threads.**

(Résultats dépendants de votre configuration machine+OS)

Au début de l'exécution les états sont :

principal : Running
thread0 : Unstarted
thread1 : Unstarted
thread2 : Unstarted

Seul le thread principal est en état d'exécution, les 3 autres nonencore démarrés.

Après invocation de la méthode **Start** de chaque thread, la tour revient au thread principal pour exécuter la boucle **for** (**int** i = 1; i < 100; i++), les 3 autres threads sont dans la file d'attente :

principal : Running
thread0 : WaitSleepJoin
thread1 : WaitSleepJoin
thread2 : WaitSleepJoin

Après la fin de l'exécution de la boucle **for** (**int** i = 1; i < 100; i++) du thread principal celui-ci est sur le point de s'arrêter, les thread1 et thread2 ont fini leur exécution, le thread0 continue son exécution car ralenti par le Sleep(2) à chaque tour de boucle :

principal : Running
thread0 : WaitSleepJoin
thread1 : Stopped
thread2 : Stopped

Après la terminaison du décompte de la boucle du thread0 jusqu'à la valeur 99, le thread0 se termine et le thread principal est sur le point de se terminer (il ne lui reste plus la dernière

instruction d'affichage " System.Console.WriteLine("fin de tous les threads.")" à exécuter) :

principal : Running
thread0 : Stopped
thread1 : Stopped
thread2 : Stopped
fin de tous les threads.

Interrompre-réveiller : Méthode Interrupt (…)

Si le thread est dans l'état **WaitSleepJoin** c'est à dire soit endormi (**Sleep**), soit dans la file d'attente attendant son tour (**Wait**), soit en attente de la fin d'un autre thread (**Join**), il est alors possible d'interrompre son état d'attente grâce à la méthode **Interrupt**. Cette méthode interrompt temporairement le thread qui l'invoque et lance une exception du type **ThreadInterruptedException**.

Dans le programme précédent nous ajoutons l'instruction "thread0.Interrupt()" dans la méthode Main, juste après la fin de la boucle **for** (**int** i = 1; i < 100; i++) {…}. Lors de l'exécution, dès que le thread0 se met en mode **WaitSleepJoin** par invocation de la méthode Sleep(2), il est interrompu :

```
public static void run0()
{
    for (int i1 = 1; i1 < 100; i1++)
    {
        System.Console.WriteLine(".... thread0 = " + i1);
        Thread.Sleep(2);
    }
}
```

ThreadInterruptedException was unhandled

Thread interrompu à partir d'un état d'attente.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

Quick Console

```
 i = 93
***  thread2 = 97
.... thread0 = 84
>>>  thread1 = 98
 i = 94
***  thread2 = 98
>>>  thread1 = 99
.... thread0 = 85
 i = 95
***  thread2 = 99
 i = 96
.... thread0 = 86
 i = 97
 i = 98
.... thread0 = 87
 i = 99
principal : Running
thread0 : Running
```

Une exception **ThreadInterruptedException** a bien été lancée et peut être interceptée dans le thread principal.

Nous listons ci-dessous le code source de la méthode Main produisant l'interruption du thread0

figurée précédemment :

```
static void Main(string[] args)
{
        Console.WriteLine("Lancement des threads :");
        AfficherDonnees obj = new AfficherDonnees();
        Thread principal = Thread.CurrentThread;
        principal.Name = "principal";
        Thread thread0 = new Thread(new ThreadStart(run0));
        thread0.Name = "thread0";
        Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
        thread1.Name = "thread1";
        Thread thread2 = new Thread(new ThreadStart(obj.run2));
        thread2.Name = "thread2";
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Start();
        thread1.Start();
        thread2.Start();
        etatsThreads(principal, thread0, thread1, thread2);
        for (int i = 1; i < 100; i++)
        {
           System.Console.WriteLine(" i = " + i);
        }
        thread0.Interrupt();
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Join();
        etatsThreads(principal, thread0, thread1, thread2);
        System.Console.WriteLine("fin de tous les threads.");
}
```

Attention : La rapidité du processeur, l'influence du CLR et la charge instantanée du système changent considérablement les résultats obtenus ! Il faut donc n'utiliser ces outils que pour du parallélisme réel et non pour du séquentiel (cf. la notion de synchronisation paragraphe suivant)

3.3 Exclusion mutuelle, concurrence, section critique, et synchronisation

ressource partagée

D'un point de vue général, plusieurs processus peuvent accéder en lecture et en écriture à un **même espace mémoire** par exemple : accéder à un spooler d'imprimante, réserver une page en mémoire centrale, etc…Cet espace mémoire partagé par plusieurs processus se dénomme **une ressource partagée**.

concurrence

Lorsque le résultat final après exécution des processus à l'intérieur sur une ressource partagée n'est pas déterministe, mais dépend de l'ordre dans lequel le système d'exploitation a procédé à l'exécution de chacun des processus, on dit que l'on est en situation de **concurrence.**

synchronisation

Lorsqu'une structure de données est accédée par des processus en situation de concurrence, il est impossible de prévoir le comportement des threads sur cette structure. Si l'on veut obtenir un comportement déterministe, il faut ordonner les exécutions des processus d'une manière séquentielle afin d'être sûr qu'**un seul** processus accède à toute la structure **jusqu'à la fin** de son exécution , puis laisse la main au processus suivant etc. Cet régulation des exécutions des processus s'appelle la **synchronisation**.

Exclusion mutuelle

> Lorsque plusieurs processus travaillent sur une ressource partagée, la synchronisation entre les divers processus sous-entend que cette ressource partagée est exclusivement à la disposition d'un processus pendant sa durée complète d'exécution. Le mécanisme qui permet à un seul processus de s'exécuter sur une ressource partagée à l'exclusion de tout autre, se dénomme l'**exclusion mutuelle**.

section critique

> Lorsqu'un bloc de lignes de code traite d'accès par threads synchronisés à une ressource partagée on dénomme ce bloc de code une **section critique**.

Tout ce que nous venons de voir sur les processus se reporte intégralement aux **threads** qui sont des processus légers.

Dans un système d'exploitation de multiprogrammation, ces situations de concurrence sont très courantes et depuis les années 60, les informaticiens ont mis en œuvre un arsenal de réponses d'exclusion mutuelle ; ces réponses sont fondées sur les notions de **verrous**, **sémaphores**, **mutex**, **moniteurs**.

Le développeur peut avoir besoin dans ses programmes de gérer des situations de concurrence comme par exemple dans un programme de réservation de place de train et d'édition de billet de transport voyageur. Le multi-threading et les outils de synchronisation que le langage de programmation fournira seront une aide très précieuse au développeur dans ce style de programmation.

Pour programmer de la synchronisation entre threads, le langage C# met à la disposition du développeur les notions de **verrous**, **sémaphores**, **mutex**, **moniteurs**.

3.4 Section critique en C# : lock

L'instruction lock (…) { …. }

Le mot clef **lock** détermine un bloc d'instructions en tant que **section critique**. Le verrouillage de cette section critique est obtenu par exclusion mutuelle sur un objet spécifique nommé verrou qui peut être dans deux états : soit disponible ou libre, soit vérouillé. Ci-dessous la syntaxe C# de l'instruction lock :

```
object verrou = new object();

lock ( verrou )
{
   … lignes de code de la section critique
}
```

Lorsqu'un thread Th1 veut entrer dans la section critique délimitée par lock, nous sommes en face de deux possibilités selon que l'objet verrou est libre ou non :

- **Si l'objet verrou est libre** (c'est à dire qu'aucun autre thread n'exécute le code de la section critique) alors on dit que le thread Th1 acquiert le verrou, d'autre part il le verrouille pour tout autre thread. Dès que le thread Th1 finit d'exécuter la dernière instruction de la section

critique, il libère le verrou qui devient disponible pour un autre thread.

- **Si l'objet verrou n'est pas libre** et qu'un thread Th2 demande à acquérir ce verrou (veut entrer dans la section critique) pendant que Th1 est dans la section critique, le thread Th2 est alors mis dans la file d'attente associée à l'objet verrou par le CLR (le thread est donc bloqué en attente). Chaque nouveau thread demandant à acquérir ce verrou est rangé dans la file d'attente du verrou tant que ce dernier n'est pas libéré. Dès que le verrou devient libre le CLR autorise le thread en tête de file à acquérir le verrou et ce thread est retiré de la file d'attente des threads du verrou.

- **Si une exception est levée** dans la section critique, le verrou est automatiquement libéré par l'instruction lock pour ce thread.

Exemple C# de synchronisation avec lock :

1°) Partons d'un exemple où le fait qu'il n'y ait pas de synchronisation provoque un comportement erratique. Soit un tableau d'entiers "**int**[ ] datas = **new int**[50]" tous à la valeur 1, nous construisons trois threads modifiant ce tableau, le premier rajoute 1 à chaque cellule du tableau, le second multiplie par 2 le contenu de chaque cellule du tableau, le troisième soustrait 1 à chaque cellule du tableau.

Nous créons une classe ThreadModifierDonnees dont la vocation est de permettre à des délégués, à qui nous donnerons un nom lors de leur instanciation, de travailler sur le tableau d'entiers à travers une méthode run().

Dans cette classe ThreadModifierDonnees la méthode run( ) effectue une action différente selon le nom du délégué qui l'invoque (ajouter 1, multiplier par 2, soustraire 1).

Afin de souligner la concurrence entre les 3 threads nous déséquilibrons les temps alloués aux threads par un endormissement différent pour le thread1 (Thread.Sleep(1);) et pour le thread2 (Thread.Sleep(0) ;), le thread3 restant indemne de toute modification temporelle :

```
public class ThreadModifierDonnees
{
    private int[ ] donnees;
    private string nom;
    public ThreadModifierDonnees(int[ ] donnees, string nom)
    {
       this.donnees = donnees;
       this.nom = nom;
    }
    public void run( )
    {
       for (int i = 0; i < donnees.Length; i++)
       {
            if (nom == "modif1")
            {
               donnees[i] += 1;
               Thread.Sleep(1);
            }
            else
               if (nom == "modif2")
               {
                  donnees[i] *= 2;
                  Thread.Sleep(0);
               }
               else
```

```
                    if (nom == "modif3")
                        donnees[i] -= 1;
            }
        }
    }

    public class Program
    {
        public static int[ ] datas = new int[50];

        public static void afficherDatas()
        {
            for (int i = 0; i < datas.Length; i++)
                Console.Write(datas[i]);
            Console.WriteLine();
        }
        public static void initDatas()
        {
            for (int i = 0; i < 50; i++)
                datas[i]=1;
        }
        static void Main(string[] args)
        {
            initDatas();
            ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas,"modif1");
            ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas,"modif2");
            ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
            afficherDatas();
            Thread thread1 = new Thread(new ThreadStart(modif_1.run));
            Thread thread2 = new Thread(new ThreadStart(modif_2.run));
            Thread thread3 = new Thread(new ThreadStart(modif_3.run));
            thread1.Start();
            thread2.Start();
            thread3.Start();
            afficherDatas();
            System.Console.ReadLine();
        }
    }
```

La méthode initDatas() rempli de 1 le tableau datas, la méthode afficherDatas affiche le contenu des 50 cellules du tableau datas.

Après avoir lancé les 3 threads nous affichons afficherDatas affiche de nouveau le contenu des 50 cellules du tableau datas. En relançant plusieurs fois le programme nous obtenons des résultats différents (qui peuvent aussi différer d'une machine à l'autre) dûs au fait que les threads ne sont pas synchronisés sur la boucle **for** de la méthode **run** :

Exécution : ***(Résultats dépendants de votre configuration machine+OS)***
11111111111111111111111111111111111111111111111111
33222222211111111111111111111111111111111111111111

Autre exécution : ***(Résultats dépendants de votre configuration machine+OS)***
11111111111111111111111111111111111111111111111111
33222222222211111111111111111111111111111111111111

Autre exécution : ***(Résultats dépendants de votre configuration machine+OS)***
11111111111111111111111111111111111111111111111111
33222221111111111111111111111111111111111111111111
etc …

2°) A l'évidence, les résultats ne sont pas fiables puisqu'ils dépendent du temps et de la charge du système, donc ce programme n'est pas valide. Afin de remédier à cet inconvénient majeur nous utilisons un verouillage de la boucle for dans la méthode run et nous déclarons un verrou de type object dans la classe ThreadModifierDonnees. L'objet verrou est **static** afin que ce soit le même objet pour toutes les instances de telle sorte que les threads consulteront le même verrou :

```
public class ThreadModifierDonnees
{
    private int[] donnees;
    private string nom;
    private static object verrou = new object();

    public ThreadModifierDonnees(int[] donnees, string nom)
    {
        this.donnees = donnees;
        this.nom = nom;
    }
    public void run()
    {
        lock (verrou)
        {
            for (int i = 0; i < donnees.Length; i++)
            {
                if (nom == "modif1")
                {
                    donnees[i] += 1;
                    Thread.Sleep(1);
                }
                else
                    if (nom == "modif2")
                    {
                        donnees[i] *= 2;
                        Thread.Sleep(0);
                    }
                    else
                        if (nom == "modif3")
                        {
                            donnees[i] -= 1;
                        }
            }
        }
    }
}
```

Le code de la classe Program reste identique, et les résultats de l'exécution sont déterministes et dirigés uniquement par l'ordre séquentiel de lancement des 3 threads

1°) Pour l'ordre de lancement des threads suivant :

```
thread1.Start();
thread2.Start();
thread3.Start();
```

Résultats d'exécution pour l'ordre de lancement 1,2,3 :
111111111111111111111111111111111111111111111111
333333333333333333333333333333333333333333333333

2°) Pour l'ordre de lancement des threads suivant:

```
thread2.Start();
thread1.Start();
thread3.Start();
```

Résultats d'exécution pour l'ordre de lancement 2,1,3 :
111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222

3°) Pour l'ordre de lancement des threads suivant:

```
thread3.Start();
thread2.Start();
thread1.Start();
```

Résultats d'exécution pour l'ordre de lancement 3,2,1 :
111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111

Si le thread prinicipal « va trop vite » on lui demande d'attendre la fin de l'exécution des 3 autres threads avant qu'il exécute la méthode afficherDatas()

```
thread1.Join(); // on attend que ce thread ait terminé la totalité de son exécution
thread2.Join(); // on attend que ce thread ait terminé la totalité de son exécution
thread3.Join(); // on attend que ce thread ait terminé la totalité de son exécution
afficherDatas();
```

3.5 Section critique en C# : Monitor

La classe Monitor

Cette classe fournit un moyen général de synchroniser les accès à un objet en C#. Elle est abstraite

> **public sealed abstract class** Monitor

Elle contient des méthodes permettant à un thread de travailler plus finement sur une section critique que l'instruction lock. Voici trois méthodes static de la classe Monitor permettant à un thread :

- D'acquérir un verrou : **public static void** Enter(object obj);
- De libérer un verrou : **public static void** Exit(object obj);
- D'essayer d'aquérir un verrou : **public static bool** TryEnter(object obj);

Les méthodes Enter et Exit permettent de délimiter une section critique :

```
object verrou = new object();
Monitor.Enter ( verrou ) ;
  … lignes de code de la section critique
 Monitor.Exit ( verrou ) ;
```

La méthode TryEnter permet à un thread de consulter l'état d'un verrou et dans l'éventualité où ce verrou est verrouillé, si le développeur le souhaite, de ne pas mettre immédiatement le thread dans

la file d'attente du verrou, mais d'effectuer d'autres actions. Ci-dessous un pseudo-code C# dans lequel le thread qui exécute la méthode2() teste l'état du verrou avant de rentrer dans la section critique contenue dans la méthode1() ou bien effectue d'autres actions si cette section critique est occupée par un autre thread :

```
private object verrou = new object();

public void methode1()
{
  Monitor.Enter ( verrou ) ;
   … lignes de code de la section critique
  Monitor.Exit ( verrou ) ;
}
…….
public void methode2()
{
  if (Monitor.TryEnter ( verrou )
    methode1( );
  else
    …Autres actions
}
```

Dans le cas où une exception serait levée dans une section critique le verrou n'est pas automatiquement levé comme avec l'instruction **lock**(){…}, Microsoft conseille au développeur de protéger son code par un **try**…**finally**. Voici le code minimal permettant d'assurer cette protection semblablement à un lock :

```
object verrou = new object();
Monitor.Enter ( verrou ) ;
try
 {
  … lignes de code de la section critique
 }
finally
 {
   Monitor.Exit ( verrou ) ;
 }
```

Dans l'exemple précédent de section critique mettant à jour les 50 cellules d'un tableau d'entiers, nous remplaçons l'instruction **lock** par un appel aux méthodes **static** *Enter* et *Exit* de la classe Monitor sans protection du code :

```
public class ThreadModifierDonnees
  {
      private int[] donnees;
      private string nom;
      private static object verrou = new object();

      public ThreadModifierDonnees(int[] donnees, string nom)
      {
         this.donnees = donnees;
         this.nom = nom;
```

```
    }
    public void run()
    {
        Monitor.Enter ( verrou ) ;
        for (int i = 0; i < donnees.Length; i++)
        {
            if (nom == "modif1")
            {
                donnees[i] += 1;
                Thread.Sleep(1);
            }
            else
                if (nom == "modif2")
                {
                    donnees[i] *= 2;
                    Thread.Sleep(0);
                }
                else
                    if (nom == "modif3")
                    {
                        donnees[i] -= 1;
                    }
        }
        Monitor.Exit ( verrou ) ;
    }
}
```

Même programme avec appel aux méthodes **static** *Enter* et *Exit* de la classe Monitor et protection du code par **try**…**finally** dans la méthode run( ):

```
    public void run()    {
      Monitor.Enter ( verrou ) ;
       try
       {
        for (int i = 0; i < donnees.Length; i++)    {
            if (nom == "modif1")
            {
                donnees[i] += 1;
                Thread.Sleep(1);
            }
            else
                if (nom == "modif2")
                {
                    donnees[i] *= 2;
                    Thread.Sleep(0);
                }
                else
                    if (nom == "modif3")
                    {
                        donnees[i] -= 1;
                    }
        }
       }
       finally   {  Monitor.Exit ( verrou ) ;   }
    }
```

3.6 Synchronisation commune aux processsus et aux threads

Il existe dans la version C# 2.0 deux classes permettant de travailler aussi bien avec des threads qu'avec des processus.

La classe Semaphore

> **public sealed class** Semaphore : WaitHandle

Semaphore est une classe dédiée à l'accès à une ressource partagée non pas par un seul thread mais par un nombre déterminé de threads lors de la création du sémaphore à travers son constructeur dont nous donnons une surcharge :

> **public** Semaphore (**int** initialCount, **int** maximumCount);
>
> initialCount = le nombre de thread autorisés à posséder le sémaphore en plus du thread principal, la valeur 0 indique que seul le thread principal possède le sépmaphore au départ.
>
> MaximumCount = le nombre maximal de thread autorisés à posséder le sémaphore.

Pour essayer de simuler une section critique accessible par un seul thread à la fois par la notion de verrou, il faut instancier un sémaphore avec un maximumCount = 1 et utiliser les méthodes *WaitOne* et *Release* de la classe Semaphore :

```
Semaphore verrou = new Semaphore (0, 1);

verrou.WaitOne ( );
… lignes de code de la section critique
verrou.Release ( );
```

Toutefois l'utilisation d'un sémaphore même avec *maximumCount = 1* ne permet pas d'ordonnancer les accès à la section critique, dans ce cas le sémaphore sert seulement à assurer qu'une section critique est accédée entièrement par un seul thread. Illustrons ce propos avec notre exemple de code partagé mettant à jour les 50 cellules d'un tableau d'entiers.

Nous créons un sémaphore **public static** dans la classe ThreadModifierDonnees que nous nommons verrou, nous supprimons les endormissements Sleep(…), nous instancions le sémaphore Semaphore verrou = **new** Semaphore (0, 1), enfin nous encadrons la section critique par les appels des méthodes *WaitOne* et *Release* :

```
public class ThreadModifierDonnees
  {
      private int[] donnees;
      private string nom;
      public static Semaphore verrou = new Semaphore ( 0, 1 );

      public ThreadModifierDonnees(int[] donnees, string nom)
      {
         this.donnees = donnees;
         this.nom = nom;
      }
      public void run()
      {
```

```
        verrou.WaitOne( );
          for (int i = 0; i < donnees.Length; i++)
          {
            if (nom == "modif1")
            {
              donnees[i] += 1;
              Thread.Sleep(1);
            }
            else
              if (nom == "modif2")
              {
                donnees[i] *= 2;
                Thread.Sleep(0);
              }
              else
                if (nom == "modif3")
                {
                  donnees[i] -= 1;
                }
          }
        verrou.Release( );
    }
}
```

Dans la méthode Main de la classe Program, par construction du sémaphore c'est le thread principal qui possède le sémaphore verrou, on fait libérer ce sémaphore par le thread principal par l'instruction "ThreadModifierDonnees.verrou.Release(1)" qui a vocation à autoriser un des 3 threads thread1, thread2 ou thread3 à entrer dans la section critique :

```
static void Main(string[] args)
{
        initDatas();
        ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas,"modif1");
        ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas,"modif2");
        ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
        afficherDatas();
        Thread thread1 = new Thread(new ThreadStart(modif_1.run));
        Thread thread2 = new Thread(new ThreadStart(modif_2.run));
        Thread thread3 = new Thread(new ThreadStart(modif_3.run));
        thread1.Start();
        thread2.Start();
        thread3.Start();
        ThreadModifierDonnees.verrou.Release ( 1 );
        afficherDatas();
        System.Console.ReadLine();
    }
}
```

Le résultat obtenu est identique à celui que nous avions obtenu avec lock ou Monitor :

Pour l'ordre de lancement des threads suivant :

```
thread1.Start();
thread2.Start();
thread3.Start();
```

Résultats d'exécution :
111111111111111111111111111111111111111111111111
333333333333333333333333333333333333333333333333

Il ne faut pas s'y méprendre, si l'on change l'ordre de lancement on a toujours l'accès garanti par le sémaphore à la section critique par un seul thread, mais l'ordre d'accès séquentiel n'est pas garanti :

1°) Pour l'ordre de lancement des threads suivant:

```
thread2.Start();
thread1.Start();
thread3.Start();
```

Résultats d'exécution :
111111111111111111111111111111111111111111111111
333333333333333333333333333333333333333333333333

2°) Pour l'ordre de lancement des threads suivant:

```
thread3.Start();
thread2.Start();
thread1.Start();
```

Résultats d'exécution :
111111111111111111111111111111111111111111111111
222222222222222222222222222222222222222222222222

La classe Mutex

> **public sealed class** Mutex : WaitHandle

Mutex est une classe dédiée à la synchronisation entre processus ou entre threads. Elle permet un accès exclusif à une ressource partagée en situation de concurrence : elle autorise un accès exclusif à cette ressource par un seul thread ou processus. Si un thread ou processus acquiert un mutex, l'autre thread ou processus qui veut acquérir ce mutex est interrompu jusqu'à ce que le premier thread ou processus libère le mutex. Elle fonctionne comme un sémaphore à un seul thread autorisé.

Les méthodes intéressantes pour gérer une section critique sont semblables à celle de la classe Semaphore, on peut considérer la classe Mutex comme une simplification de la classe Semaphore dans le cas d'un seul thread ou d'un seul processus qui accède à la section critique. Elle possède toutefois l'avantage sur la classe Semaphore de permettre de gérer l'**ordre d'accès séquentiel à la section critique.**

On utilise les méthodes *WaitOne* et *ReleaseMutex* de la classe Mutex:

```
Semaphore verrou = new Mutex ( );

verrou.WaitOne ( );
… lignes de code de la section critique
verrou.ReleaseMutex ( );
```

Ci-dessous le code de la classe ThreadModifierDonnees avec un mutex **public** que nous nommons verrou, nous avons aussi supprimé les endormissements Sleep(…), et nous encadrons la section critique par les appels des méthodes *WaitOne* et *ReleaseMutex* :

```
public class ThreadModifierDonnees
  {
      private int[] donnees;
      private string nom;
      public static Mutex verrou = new Mutex ( );

      public ThreadModifierDonnees(int[] donnees, string nom)
      {
         this.donnees = donnees;
         this.nom = nom;
      }
      public void run()
      {
         verrou.WaitOne( );
            for (int i = 0; i < donnees.Length; i++)
            {
               if (nom == "modif1")
               {
                  donnees[i] += 1;
                  Thread.Sleep(1);
               }
               else
                  if (nom == "modif2")
                  {
                     donnees[i] *= 2;
                     Thread.Sleep(0);
                  }
                  else
                     if (nom == "modif3")
                     {
                        donnees[i] -= 1;
                     }
            }
         verrou.ReleaseMutex( );
      }
  }
```

Nous retrouvons dans la méthode Main de la classe Program, un code identique à celui correspondant à une utilisation d'un **lock** ou d'un Monitor :

```
  static void Main(string[] args)
  {
        initDatas();
        ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas,"modif1");
        ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas,"modif2");
        ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
        afficherDatas();
        Thread thread1 = new Thread(new ThreadStart(modif_1.run));
        Thread thread2 = new Thread(new ThreadStart(modif_2.run));
        Thread thread3 = new Thread(new ThreadStart(modif_3.run));
        thread1.Start();
        thread2.Start();
        thread3.Start();
        afficherDatas();
        System.Console.ReadLine();
     }
  }
```

Nous donnons les résultats obtenus selon l'odre de lancement des thread :

1°) Pour l'ordre de lancement des threads suivant :

```
thread1.Start();
thread2.Start();
thread3.Start();
```

Résultats d'exécution :
111111111111111111111111111111111111111111111111
333333333333333333333333333333333333333333333333

2°) Pour l'ordre de lancement des threads suivant:

```
thread2.Start();
thread1.Start();
thread3.Start();
```

Résultats d'exécution :
111111111111111111111111111111111111111111111111
222222222222222222222222222222222222222222222222

3°) Pour l'ordre de lancement des threads suivant:

```
thread3.Start();
thread2.Start();
thread1.Start();
```

Résultats d'exécution :
111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111

Ces résultats sont identiques à ceux que l'on obtient en utilisant un lock.

En conclusion, l'instruction **lock** s'avère être la plus souple et la plus simple à utiliser pour définir une section critique et pour assurer une synchronisation efficace des threads sur cette section critique.

IHM,Winforms, exceptions, flux

- Programmation événementielle
- Les événements
- Propriétés et indexeurs
- Fenêtres et ressources
- Contrôles dans les formulaires
- Exceptions comparées
- Flux et fichiers : données simples

Programmation événementielle et visuelle

Plan du chapitre:

Introduction

1. Programmation visuelle basée sur les pictogrammes

2. Programmation orientée événements

3. Normalisation du graphe événementiel

le graphe événementiel *arcs et sommets*
les diagrammes d'états UML *réduits*

4. Tableau des actions événementielles

5. Interfaces liées à un graphe événementiel

6. Avantages et modèle de développement RAD visuel

le modèle de la spirale *(B.Boehm)*
le modèle incrémental

1. Programmation visuelle basée sur les pictogrammes

Le développement visuel rapide d'application est fondé sur le concept de programmation visuelle associée à la montée en puissance de l'utilisation des Interactions Homme-Machine (IHM) dont le dynamisme récent ne peut pas être méconnu surtout par le débutant. En informatique les systèmes MacOs, Windows, les navigateurs Web, sont les principaux acteurs de l'ingénierie de l'IHM. Actuellement dans le développement d'un logiciel, un temps très important est consacré à l'ergonomie et la communication, cette part ne pourra que grandir dans un avenir proche; car les utilisateurs veulent s'adresser à des logiciels efficaces (ce qui va de soi) mais aussi conviviaux et faciles d'accès.

Les développeurs ont donc besoin d'avoir à leur disposition des produits de développement adaptés aux nécessités du moment. A ce jour la programmation visuelle est une des réponses à cette attente des développeurs.

La programmation visuelle au tout début a été conçue pour des personnes n'étant pas des programmeurs en basant ses outils sur des manipulations de pictogrammes.

Le raisonnement communément admis est qu'un dessin associé à une action élémentaire est plus porteur de sens qu'une phrase de texte.

A titre d'exemple ci-dessous l'on enlève le "fichier.bmp" afin de l'effacer selon deux modes de communication avec la machine: utilisation d'icônes ou entrée d'une commande textuelle.

Effacement avec un langage d'action visuelle (souris)

Action :	**Réponse :**

Effacement avec un langage textuel (clavier)

Action :	**Réponse :**
del c:\Exemple\Fichier.bmp	| ?

Nous remarquons donc déjà que l'interface de communication MacOs, Windows dénommée "bureau électronique" est en fait un outil de programmation de commandes systèmes.

Un langage de programmation visuelle permet "d'écrire" la partie communication d'un programme uniquement avec des dessins, diagrammes, icônes etc... Nous nous intéressons aux systèmes RAD (Rapid Application Development) visuels, qui sont fondés sur des langages objets à bases d'icônes ou pictogrammes. Visual Basic de MicroSoft est le premier RAD visuel a avoir été commercialisé dès 1991, il est fondé sur un langage Basic étendu incluant des objets étendus en VB.Net depuis 2001, puis dès 1995 Delphi le premier RAD visuel de Borland fondé sur Pascal objet, puis actuellement toujours de Borland : C++Builder RAD visuel fondé sur le langage C++ et Jbuilder, NetBeans RAD visuel de Sun fondés sur le langage Java, Visual C++, Visual J++ de Microsoft, puis Visual studio de Microsoft etc...

Le développeur trouve actuellement, une offre importante en outil de développement de RAD visuel y compris en open source. Nous proposons de définir un langage de RAD visuel ainsi :

Un **langage visuel** dans un RAD visuel est un **générateur de code source** du langage de base qui, derrière chaque action visuelle (dépôt de contrôle, click de souris, modifications des propriétés, etc...) engendre des lignes de code automatiquement et d'un manière transparente au développeur.

Des outils de développement tels que Visual Basic, Delphi ou C# sont adaptés depuis leur création à la programmation visuelle pour débutant. Toutefois l'efficacité des dernières versions a étendu leur champ au développement en général et dans l'activité industrielle et commerciale avec des versions "entreprise" pour VB et "Architect "pour Delphi et les versions .Net.

En outre, le système windows est le plus largement répandu sur les machines grand public (90% des PC vendus en sont équipés), il est donc très utile que le débutant en programmation sache utiliser un produit de développement (rapide si possible) sur ce système.

Proposition :

Nous considérons dans cet ouvrage, la programmation visuelle à la fois comme une **fin** et comme un **moyen**.

La programmation visuelle est sous-tendue par la réactivité des programmes en réponse aux actions de l'utilisateur. Il est donc nécessaire de construire des programmes qui répondent à des

sollicitations externes ou internes et non plus de programmer séquentiellement (ceci est essentiellement dû aux architectures de Von Neumann des machines) : ces sollicitations sont appelées des événements.

Le concept de **programmation dirigée ou orientée par les événements** est donc la composante essentielle de la programmation visuelle.

Terminons cette présentation par 5 remarques sur le concept de RAD :

Utiliser un RAD simple mais puissant

Nous ne considérerons pas comme utile pour des débutants de démarrer la programmtion visuelle avec des RAD basés sur le **langage C++**. Du fait de sa large permissivité ce langage permet au programmeur d'adopter certaines **attitudes dangereuses** sans contrôle possible. Seul le programmeur confirmé au courant des pièges et des subtilités de la programmation et du langage, pourra exploiter sans risque la richesse de ce type de RAD.

Avoir de bonnes méthodes dès le début

Le RAD Visual C# inclus dans Visual Studio, a des **caractéristiques très proches de celles de ses parents Java et Delphi,** tout en apportant quelques améliorations. L'aspect fortement typé du langage C# autorise la prise en compte par le développeur débutant de bonnes attitudes de programmation.

C'est Apple qui en a été le promoteur

Le premier environnement de développement visuel professionnel fut basé sur **Object Pascal** et a été conçu par Apple pour le système d'exploitation **MacOs**, sous la dénomination de **MacApp** en 1986. Cet environnement objet visuel permettait de développer des applications MacIntosh avec souris, fenêtre, menus déroulants etc...

Microsoft l'a popularisé

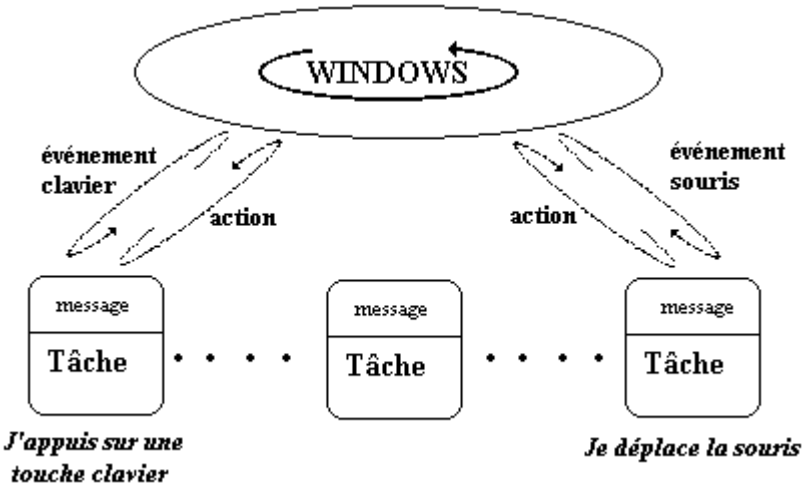
Le RAD Visual Basic de MicroSof conçu à partir de 1992, basé sur le langage Basic avait pour objectif le développement de petits logiciels sous Windows par des **programmeurs non expérimentés et occasionnels**. Actuellement il se décline en VB.Net un langage totalement orienté objet faisant partie intégrante de la plate-forme .Net Framework de Microsoft avec Visual C#.

Le concept de RAD a de beaux jours devant lui

Le métier de développeur devrait à terme, consister grâce à des outils tels que les RAD visuels, à prendre un "caddie" et à aller dans un supermarché de composants logiciels génériques adaptés à son problème. Il ne lui resterait plus qu'à assembler le flot des événements reliant entre eux ces logiciels en kit.

2. Programmation orientée événements

Sous les versions actuelles de Windows, système multi-tâches préemptif sur micro-ordinateur, les concepts quant à la programmation par événement restent sensiblement les mêmes que sous les anciennes versions.



Nous dirons que le système d'exploitation passe l'essentiel de son " temps " à attendre une action de l'utilisateur (événement). Cette action déclenche un message que le système traite et envoie éventuellement à une application donnée.

Une définition de la programmation orientée événements

Logique de conception selon laquelle un programme est construit avec des objets et leurs propriétés et d'après laquelle les interventions de l'utilisateur sur les objets du programme déclenchent l'exécution des routines associées.

Par la suite, nous allons voir dans ce chapitre que la programmation d'une application " windows-like " est essentiellement une programmation par événements associée à une programmation classique.

Nous pourrons construire un logiciel qui réagira sur les interventions de l'utilisateur si nous arrivons à intercepter dans notre application les messages que le système envoie. Or l'environnement RAD (C#, comme d'ailleurs avant lui Visual Basic de Microsoft), autorise la consultation de tels messages d'un façon simple et souple.

Deux approches pour construire un programme

- *L'**approche événementielle*** intervient principalement dans l'interface entre le logiciel et l'utilisateur, mais aussi dans la liaison dynamique du logiciel

avec le système, et enfin dans la sécurité.

- *L'**approche visuelle*** nous aide et simplifie notre tâche dans la construction du dialogue homme-machine.

- La combinaison de ces deux approches produit un logiciel habillé et adapté au système d'exploitation.

Il est possible de relier certains objets entre eux par des relations événementielles. Nous les représenterons par un graphe (structure classique utilisée pour représenter des relations).

Lorsque l'on utilise un système multi-fenêtré du genre windows, l'on dispose du clavier et de la souris pour agir sur le système. En utilisant un RAD visuel, il est possible de **construire un logiciel qui se comporte comme le système sur lequel il s'exécute**. L'intérêt est que l'utilisateur aura moins d'efforts à accomplir pour se servir du programme puisqu'il aura des fonctionnalités semblables au système. Le fait que l'utilisateur reste dans un **environnement familier** au niveau de la manipulation et du confort du dialogue, assure le logiciel d'un capital confiance de départ non négligeable.

3. Normalisation du graphe événementiel

Il n'existe que peu d'éléments accessibles aux débutants sur la programmation orientée objet par événements. Nous construisons une démarche méthodique pour le débutant, en partant de remarques simples que nous décrivons sous forme de schémas dérivés des diagrammes d'états d'UML. Ces schémas seront utiles pour nous aider à décrire et à implanter des relations événementielles en C# ou dans un autre RAD événementiel.

Voici deux principes qui pour l'instant seront suffisants à nos activités de programmation.

Dans une interface windows-like nous savons que:

- Certains événements déclenchent immédiatement des actions comme par exemple des appels de routines système.

- D'autres événements ne déclenchent pas d'actions apparentes mais activent ou désactivent certains autres événements système.

Nous allons utiliser ces deux principes pour conduire notre programmation par événements.

Nous commencerons par **le concept d'activation et de désactivation**, les autres événements fonctionneront selon les mêmes bases. Dans un enseignement sur la programmation événementielle, nous avons constaté que ce **concept était suffisant** pour que les étudiants comprennent les fondamentaux de l'approche événementielle.

Remarque :

> Attention! Il ne s'agit que d'une manière particulière de conduire notre programmation, ce ne peut donc être ni la seule, ni la meilleure (le sens accordé au mot meilleur est relatif au domaine pédagogique). Cette démarche s'est révélée être fructueuse lors d'enseignements d'initiation à ce genre de programmation.

Hypothèses de construction

Nous supposons donc que lorsque l'utilisateur intervient sur le programme en cours d'exécution, ce dernier réagira en première analyse de deux manières possibles :

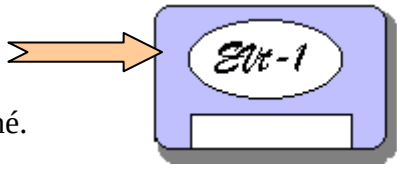
- soit il lancera l'appel d'une routine (exécution d'une action, calcul, lecture de fichier, message à un autre objet comme ouverture d'une fiche etc...),
- soit il modifiera l'état d'activation d'autres objets du programme et/ou de lui-même, soit il ne se passera rien, nous dirons alors qu'il s'agit d'une modification nulle.

Ces hypothèses sont largement suffisantes pour la plupart des logiciels que nous pouvons raisonnablement espérer construire en initiation. Les concepts plus techniques de messages dépassent assez vite l'étudiant qui risque de replonger dans de " la grande bidouille ".

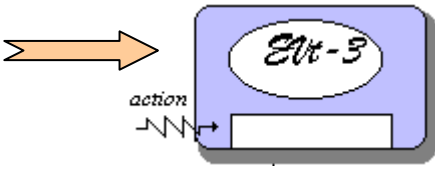
3.1 le graphe événementiel arcs et sommets

Nous proposons de construire un graphe dans lequel :

chaque **sommet** est un objet sensible à un événement donné.

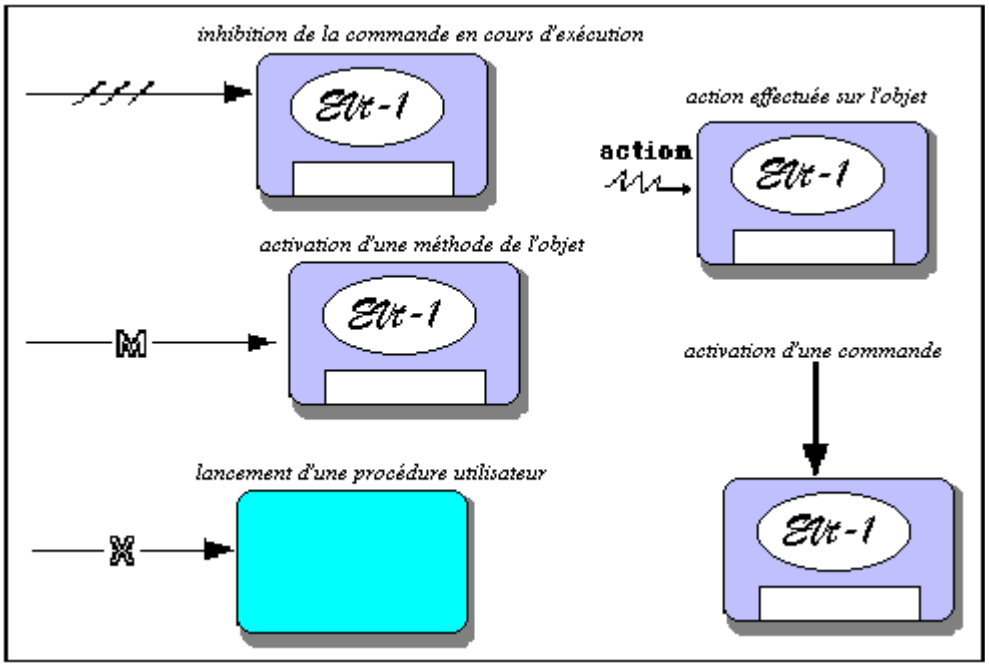


L'événement donné est déclenché par une action extérieure à l'objet.

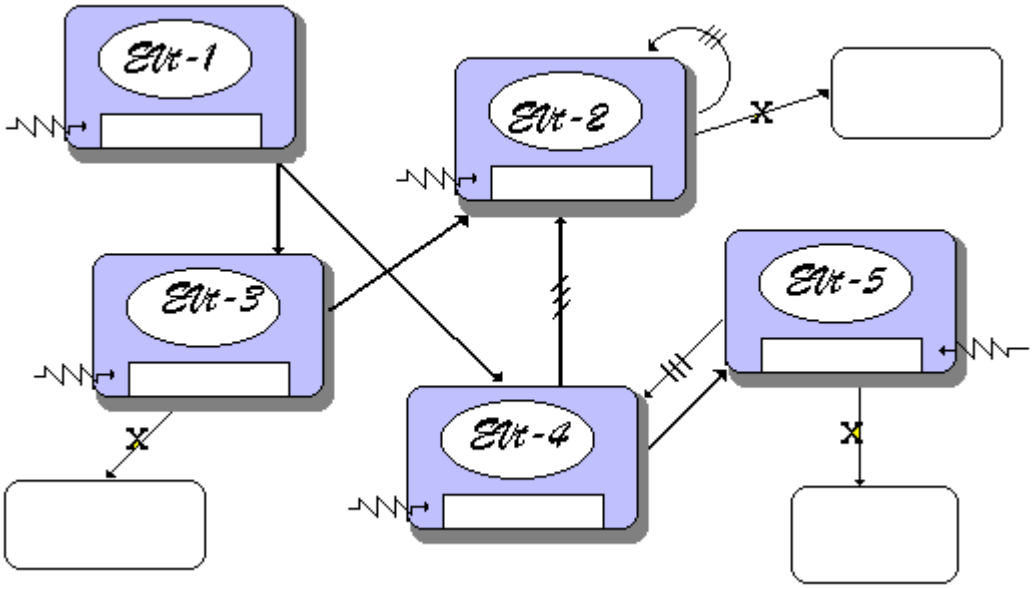


Les arcs du graphe représentent des actions lancées par un sommet.

Les actions sont de 4 types

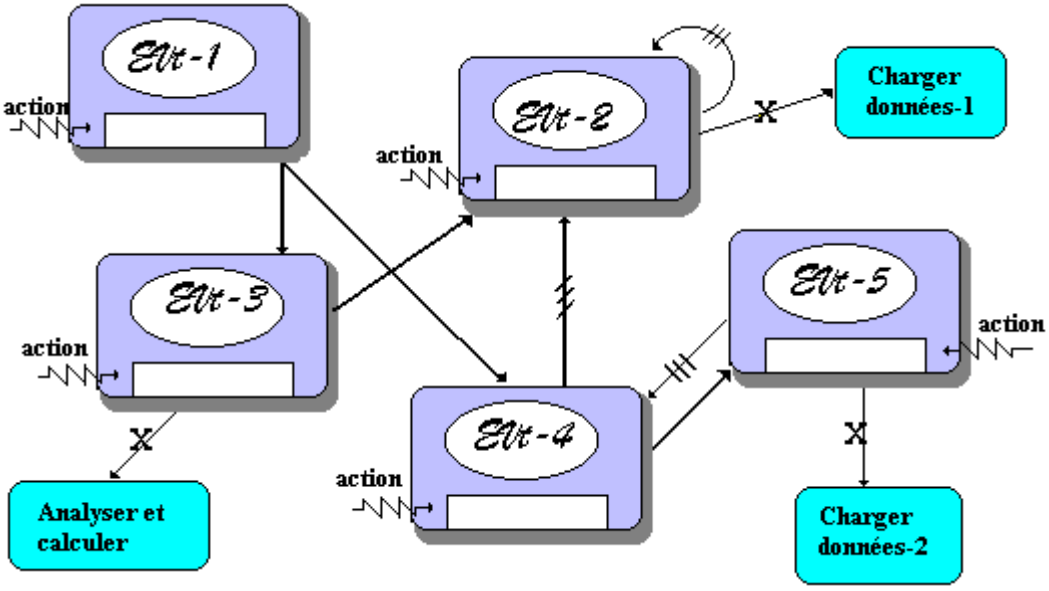


Soit le graphe événementiel suivant composé de 5 objets sensibles chacun à un événement particulier dénoté Evt-1,..., Evt-5; ce graphe comporte des réactions de chaque objet à l'événement auquel il est sensible :



Imaginons que ce graphe corresponde à une analyse de chargements de deux types différents de données à des fins de calcul sur leurs valeurs.

La figure suivante propose un tel graphe événementiel à partir du graphe vide précédent.



Cette notation de graphe événementiel est destinée à s'initier à la pratique de la description d'au maximum 4 types de réactions d'un objet sur la sollicitation d'un seul événement.

Remarques :

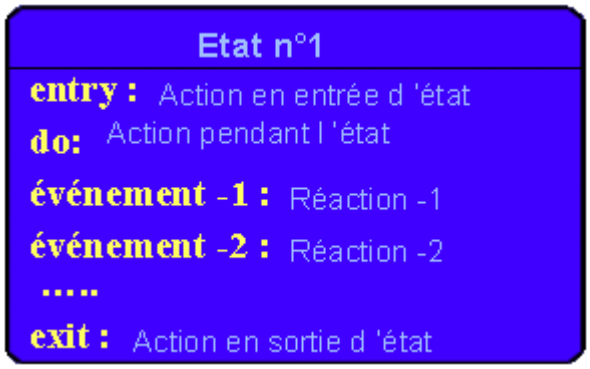
- L'arc nommé, représentant l'activation d'une méthode correspond très

exactement à la notation UML de l'envoi d'un message.

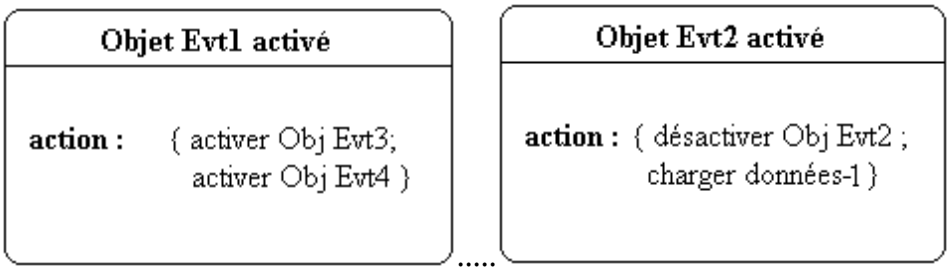
- Lorsque nous voudrons représenter d'une manière plus complète d'autres réactions d'**un seul** objet **à plusieurs événements différents**, nous pourrons utiliser la notation UML réduite de diagramme d'état pour un objet (réduite parce qu'un objet **visuel** ne prendra pour nous, que 2 états: activé ou désactivé).

3.2 les diagrammes d'états UML réduits

Nous livrons ci-dessous la notation générale de diagramme d'état en UML, les cas particuliers et les détails complets sont décrits dans le document de spécification d'UML.



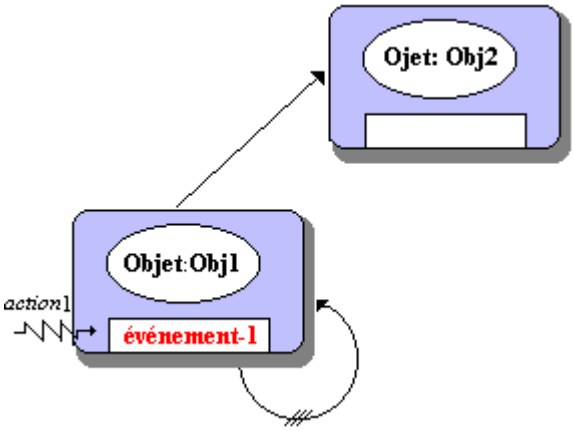
Voici les diagrammes d'états réduits extraits du graphe événementiel précédent, pour les objets **Evt-1** et **Evt-2** :



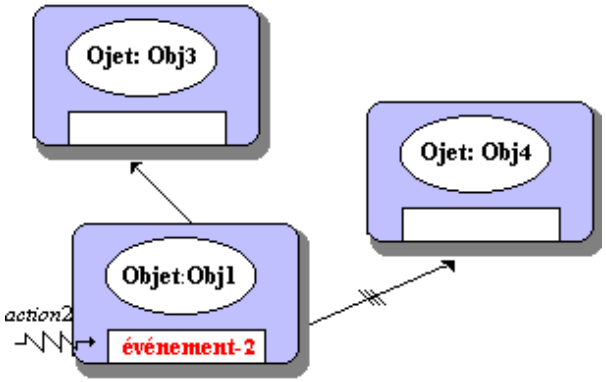
Nous remarquerons que cette écriture, pour l'instant, ne produit pas plus de sens que le graphe précédent qui comporte en sus la vision globale des interrelations entre les objets.

Ces diagrammes d'états réduits deviennent plus intéressants lorsque nous voulons exprimer le fait par exemple, qu'un seul objet **Obj1** réagit à 3 événements (événement-1, événement-2, événement-3). Dans ce cas décrivons les portions de graphe événementiel associés à chacun des événements :

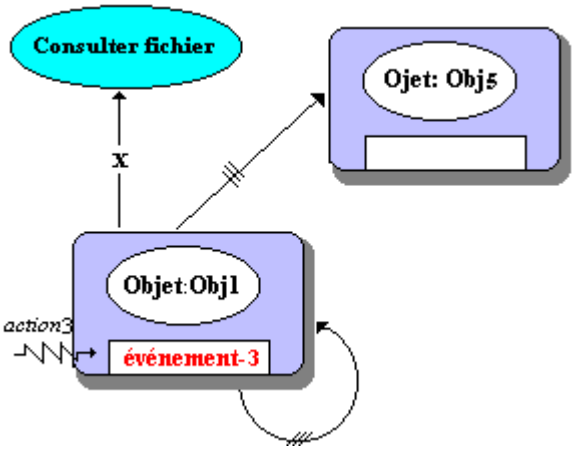
Réaction de obj1 à l'événement-1 :



Réaction de obj1 à l'événement-2 :



Réaction de obj1 à l'événement-3 :

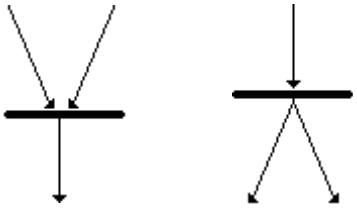


Synthétisons dans un diagramme d'état réduit les réactions à ces 3 événements :

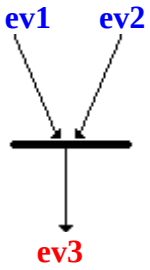
Objet:Obj activé
action.1 : { activer Obj2 ; désactiver Obj1 }
action.2 : { activer Obj3; désactiver Obj4 }
action.3 : { désactiver Obj1; activer Obj5 ; consulter fichier }

Lorsque nous jugerons nécessaire à la compréhension de relations événementielles dans un logiciel visuel, nous pourrons donc utiliser ce genre de diagramme pour renforcer la sémantique de conception des objets visuels. La notation UML sur les diagrammes d'états comprend les notions d'état de départ, de sortie, imbriqué, historisé, concurrents...

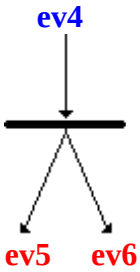
Lorsque cela sera nécessaire nous utiliserons la notation UML de synchronisation d'événements :



Dans le premier cas la notation représente la conjonction des deux événements **ev1** et **ev2** qui déclenche l'événement **ev3**.



Dans le second cas la notation représente l'événement **ev4** déclenchant conjointement les deux événements **ev5** et **ev6**.



4. Tableau des actions événementielles

L'exemple de graphe événementiel précédent correspond à une application qui serait sensible à 5 événements notés EVT-1 à EVT-5, et qui exécuterait 3 procédures utilisateur. Nous notons dans un tableau (nommé " ***tableau des actions événementielles*** ")les résultats obtenus par analyse du graphe précédent, événement par événement..

EVT-1	EVT-3 activable EVT-4 activable

EVT-2	Appel de procédure utilisateur "chargement-1" désactivation de l' événement EVT-2
EVT-3	Appel de procédure utilisateur "Analyser" EVT-2 activable
EVT-4	EVT-2 désactivé EVT-5 activable
EVT-5	EVT-4 désactivé immédiatement Appel de procédure utilisateur "chargement-2"

Nous adjoignons à ce tableau une table des états des événements dès le lancement du logiciel (elle correspond à l'état initial des objets). Par exemple ici :

Evt1	activé
Evt2	désactivé
Evt3	activé
Evt4	activé
Evt5	désactivé

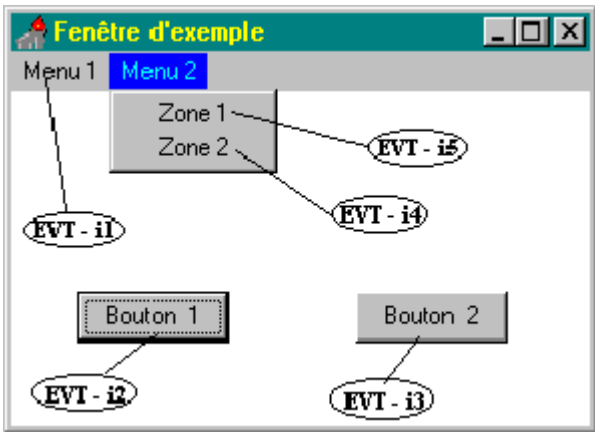
etc...

5. Interfaces liées à un graphe événementiel

construction d'interfaces liées au graphe précédent

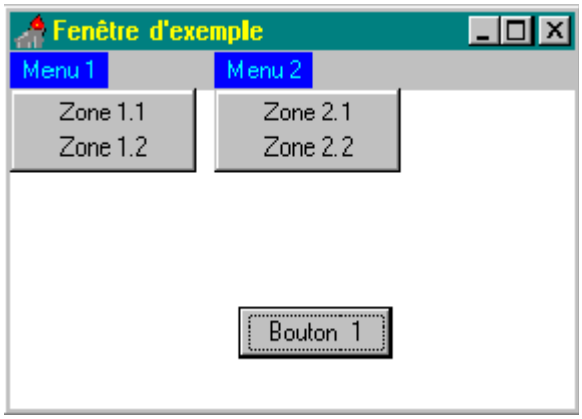
Dans l'exemple de droite, (i1,i2,i3,i4,i5)est une permutation sur (1,2,3,4,5) .

Ce qui nous donne déjà 5!=120 interfaces possibles avec ces objets et uniquement avec cette topologie.

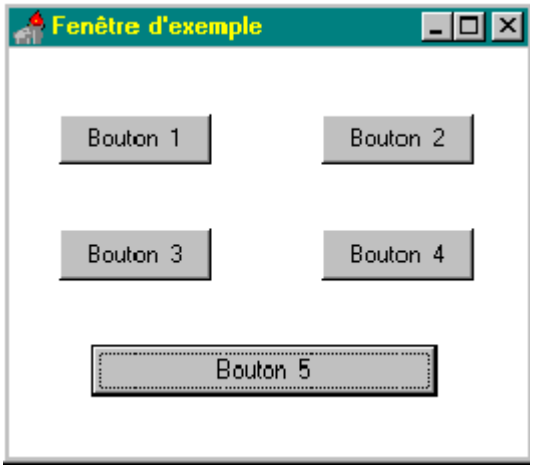


Interface n°1

La figure précédente montre une IHM à partir du graphe événementiel étudié plus haut. Ci-dessous deux autres structures d'interfaces possibles avec les mêmes objets combinés différemment et associés au même graphe événementiel :



Interface n°2



Interface n°3

Pour les choix n°2 et n°3, il y a aussi 120 interfaces possibles…

6. Avantages du modèle de développement RAD visuel

L'approche uniquement structurée (privilégiant les fonctions du logiciel) impose d'écrire du code long et compliqué en risquant de ne pas aboutir, car il faut tout tester afin d'assurer un bon fonctionnement de tous les éléments du logiciel.

L'approche événementielle préfère bâtir un logiciel fondé sur une construction graduelle en fonction des besoins de communication entre l'humain et la machine. Dans cette optique, le programmeur élabore les fonctions associées à une action de communication en privilégiant le dialogue. Ainsi les actions internes du logiciel sont subordonnées au flux du dialogue.

Avantages liés à la programmation par RAD visuel

- Il est possible de construire très rapidement un prototype.
- Les fonctionnalités de communication sont les guides principaux du développement (approche plus vivante et attrayante).
- L'étudiant est impliqué immédiatement dans le processus de conception - construction.

L'étudiant acquiert très vite comme naturelle l'attitude de réutilisation en se servant de " logiciels en kit " (soit au début des composants visuels ou non, puis par la suite ses propres composants).

Il n'y a pas de conflit ni d'incohérence avec la démarche structurée : les algorithmes étant conçus comme des boîtes noires permettant d'implanter certaines actions, seront réutilisés immédiatement.

La méthodologie objet de COO et de POO reste un facteur d'intégration général des activités du programmeur.

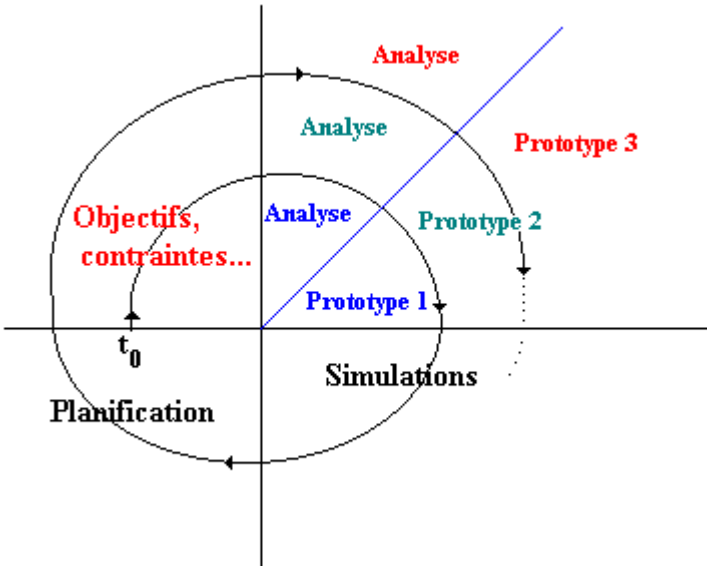
Les actions de communications classiques sont assurées immédiatement par des objets standards (composants visuels ou non) réagissant à des événements extérieurs.

Le RAD fournira des classes d'objets standards non visuels (extensibles si l'étudiant augmente sa compétence) gérant les structures de données classiques (Liste, arbre, etc..). L'extensibilité permet à l'enseignant de rajouter ses kits personnels d'objets et de les mettre à la disposition des étudiants comme des outils standards.

Modèles de développement avec un RAD visuel objet

Le développement avec ce genre de produit autorise une logique générale articulée sur la combinaison de deux modèles de développement :

le modèle de la spirale (B.Boehm) en version simplifiée

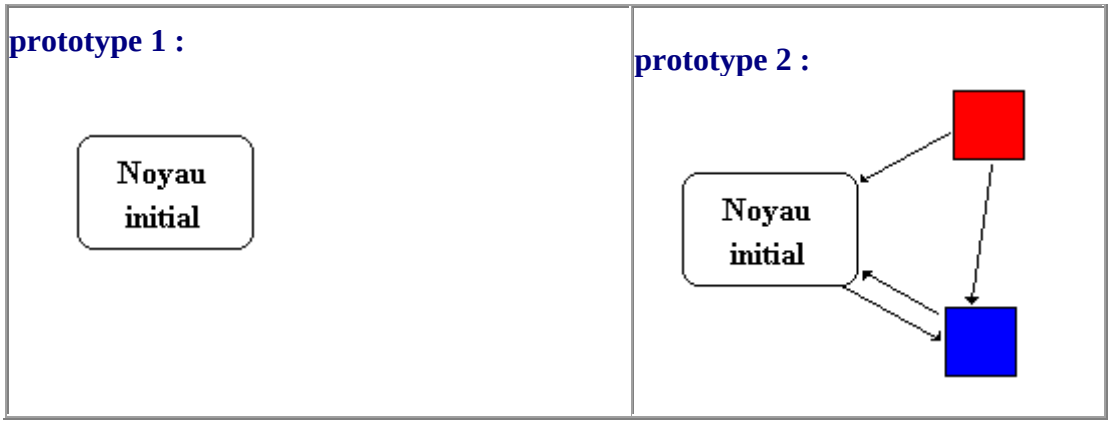


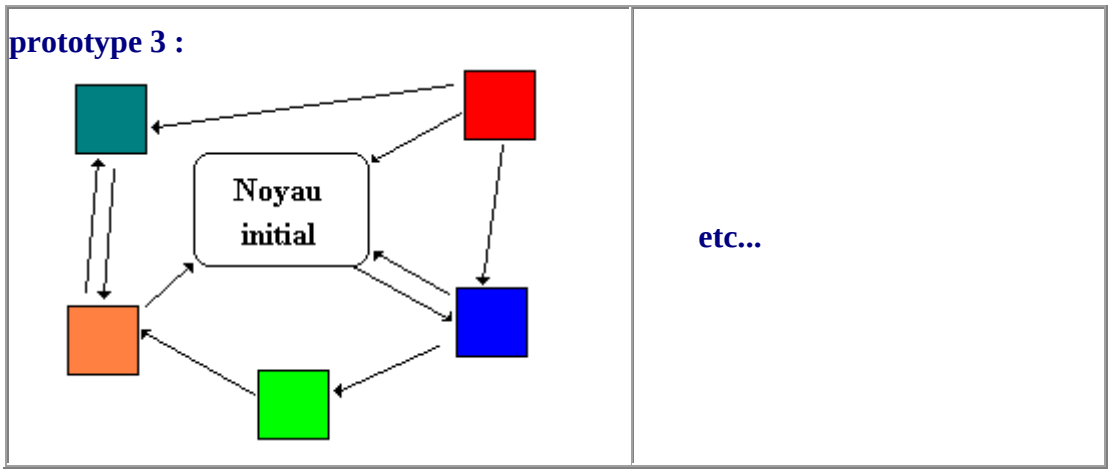
Dans le modèle de la spirale la programmation exploratoire est utilisée sous forme de prototypes simplifiés cycle après cycle. L'analyse s'améliore au cours de chaque cycle et fixe le type de développement pour ce tour de spirale.

le modèle incrémental

Il permet de réaliser chaque prototype avec un bloc central au départ, s'enrichissant à chaque phase de nouveaux composants et ainsi que de leurs interactions.

Associé à un cycle de prototypage dans la spirale, une seule famille de composants est développée pour un cycle fixé.





Ce modèle de développement à l'aide d'objets visuels ou non, fournira en fin de parcours un prototype opérationnel qui pourra s'intégrer dans un projet plus général.

Nous verrons sur des exemples comment ce type d'outil peut procurer aussi des avantages au niveau de la programmation défensive.

7. principes d'élaboration d'une IHM

Nous énumérons quelques principes utiles à l'élaboration d'une interface associée étroitement à la programmation événementielle

Notre point de vue reste celui du pédagogue et non pas du spécialiste en ergonomie ou en psychologie cognitive qui sont deux éléments essentiels dans la conception d'une interface homme-machine (**IHM**). Nous nous efforcerons d'utiliser les principes généraux des **IHM** en les mettant à la portée d'un débutant avec un double objectif :

- Faire écrire des programmes interactifs. Ce qui signifie que le programme doit communiquer avec l'utilisateur qui reste l'acteur privilégié de la communication. Les programmes sont Windows-like et nous nous servons du RAD visuel C# pour les développer. Une partie de la spécification des programmes s'effectue avec des objets graphiques représentant des classes (programmation objet visuelle).

- Le développeur peut découpler pendant la conception la programmation de son interface de la programmation des tâches internes de son logiciel (pour nous généralement ce sont des algorithmes ou des scénarios objets).

Nous nous efforçons aussi de ne proposer que des outils pratiques qui sont à notre portée et utilisables rapidement avec un système de développement RAD visuel comme C#.

Interface homme-machine, les concepts

Les spécialistes en ergonomie conceptualisent une IHM en six concepts :

- les objets d'entrée-sortie,
- les temps d'attente (temps de réponse aux sollicitations),
- le pilotage de l'utilisateur dans l'interface,
- les types d'interaction (langage,etc..) ,
- l'enchaînement des opérations,
- la résistance aux erreurs (ou robustesse qui est la qualité qu'un logiciel à fonctionner même dans des conditions anormales).

Un principe général provenant des psycho-linguistes guide notre programmeur dans la complexité informationnelle : la mémoire rapide d'un humain ne peut être sollicitée que par un nombre limité de concepts différents en même temps (nombre compris entre sept et neuf). Développons un peu plus chacun des six concepts composants une interface.

Les objets d'entrée-sortie

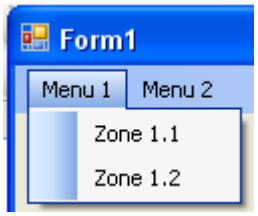
Concept

Une IHM présente à l'utilisateur un éventail d'informations qui sont de deux ordres : des commandes entraînant des actions internes sur l'IHM et des données présentées totalement ou partiellement selon l'état de l'IHM.

Les commandes participent à la " saisie de l'intention d'action" de l'utilisateur, elles sont matérialisées dans le dialogue par des ***objets d'entrée*** de l'information (boîtes de saisie, boutons, menus etc...).

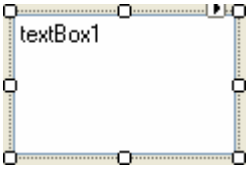
Voici avec un RAD visuel comme C#, des objets visuels associés aux objets d'entrée de l'information, ils sont très proches visuellement des objets que l'on trouve dans d'autres RAD visuels, car en fait ce sont des surcouches logiciels de contrôles de base du système d'exploitation (qui est lui-même fenêtré et se présente sous forme d'une IHM dénommée bureau électronique).

un bouton



Un menu

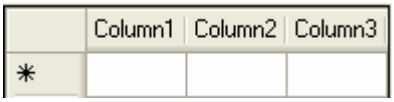
une boîte de saisie mono-ligne



une boîte de saisie multi-ligne

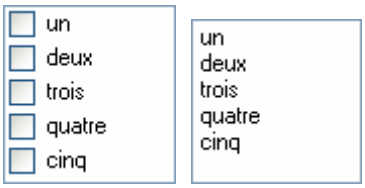
Les données sont présentées à un instant précis du dialogue à travers des ***objets de sortie*** de l'information (boîte d'édition monoligne, multiligne, tableaux, graphiques, images, sons etc...).

Ci-dessous quelques objets visuels associés à des objets de sortie de l'information :

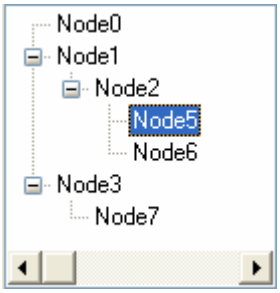


un dataGridView (table)

un pictureBoxe(image jpg, png, bm ,ico,...)



un checkedListBox et un listBox (listes)



un treeView (arbre)

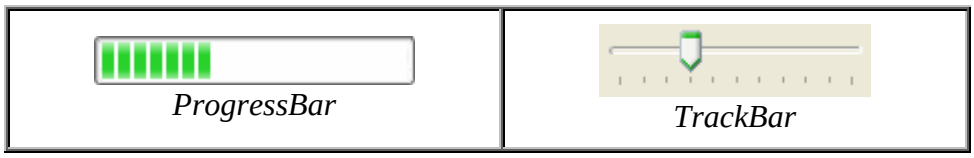
Les temps d'attente

Sur cette question, une approche psychologique est la seule réponse possible, car l'impression d'attente ne dépend que de celui qui attend selon son degré de patience. Toutefois, puisque nous avons parlé de la mémoire à court terme (mémoire rapide), nous pouvons nous baser sur les temps de persistance généralement admis (environ 5 secondes).

Nous considérerons qu'en première approximation, si le délai d'attente est :

- inférieur à environ une seconde la réponse est quasi-instantanée,
- compris entre une seconde et cinq secondes il y a attente, toutefois la mémoire rapide de l'utilisateur contient encore la finalité de l'opération en cours.
- lorsque l'on dépasse la capacité de mémorisation rapide de l'utilisateur alors il faut soutenir l'attention de l'utilisateur en lui envoyant des informations sur le déroulement de l'opération en cours (on peut utiliser pour cela par exemple des barres de défilement, des jauges, des boîtes de dialogue, etc...)

Exemples de quelques classes d'objets visuels de gestion du délai d'attente :



statusStrip (avec deux éléments ici)

Concept

Le pilotage de l'utilisateur

Nous ne cherchons pas à explorer les différentes méthodes utilisables pour piloter un utilisateur dans sa navigation dans une interface. Nous adoptons plutôt la position du concepteur de logiciel qui admet que le futur utilisateur ne se servira de son logiciel que d'une façon épisodique. Il n'est donc pas question de demander à l'utilisateur de connaître en permanence toutes les fonctionnalités du logiciel.

En outre, il ne faut pas non plus submerger l'utilisateur de conseils de guides et d'aides à profusion, car ils risqueraient de le détourner de la finalité du logiciel. Nous préférons adopter une ligne moyenne qui consiste à fournir de petites aides rapides contextuelles (au moment où l'utilisateur en a besoin) et une aide en ligne générale qu'il pourra consulter s'il le souhaite.

Ce qui revient à dire que l'on accepte deux niveaux de navigation dans un logiciel :

- le niveau de surface permettant de réagir aux principales situations,

- le niveau approfondi qui permet l'utilisation plus complète du logiciel.

Il faut admettre que le niveau de surface est celui qui restera le plus employé (l'exemple d'un logiciel de traitement de texte courant du commerce montre qu'au maximum 30% des fonctionnalités du produit sont utilisées par plus de 90% des utilisateurs).

Pour permettre un pilotage plus efficace on peut établir à l'avance un graphe d'actions possibles du futur utilisateur (nous nous servirons du graphe événementiel) et ensuite diriger l'utilisateur dans ce graphe en matérialisant (masquage ou affichage) les actions qui sont réalisables. En application des notions acquises dans les chapitres précédents, nous utiliserons un pilotage dirigé par la syntaxe comme exemple.

Les types d'interaction

Concept

Le tout premier genre d'interaction entre l'utilisateur et un logiciel est apparu sur les premiers systèmes d'exploitation sous la forme d'un langage de commande. L'utilisateur dispose d'une famille de commandes qu'il est censé connaître, le logiciel étant doté d'une interface interne (l'interpréteur de cette famille de commandes). Dès que l'utilisateur tape textuellement une commande (exemple MS-DOS " **dir c: /w** "), le système l'interprète (dans l'exemple : lister en prenant toutes les colonnes d'écran, les bibliothèques et les fichiers du disque C).

Nous adoptons comme mode d'interaction entre un utilisateur et un logiciel, une extension plus moderne de ce genre de dialogue, en y ajoutant, en privilégiant, la notion d'objets visuels permettant d'effectuer des commandes par actions et non plus seulement par syntaxe textuelle pure.

Nous construisons donc une interface tout d'abord ***essentiellement*** à partir des interactions événementielles, puis lorsque cela est utile ou nécessaire, nous pouvons ajouter un interpréteur de langage (nous pouvons par exemple utiliser des automates d'états finis pour la reconnaissance).

L'enchaînement des opérations

Concept

Nous savons que nous travaillons sur des machines de Von Neumann, donc séquentielles, les opérations internes s'effectuant selon un ordre unique sur lequel l'utilisateur n'a aucune prise. L'utilisateur est censé pouvoir agir d'une manière " aléatoire ". Afin de simuler une certaine liberté d'action de l'utilisateur nous lui ferons parcourir un graphe événementiel prévu par le programmeur. Il y a donc contradiction entre la rigidité séquentielle imposée par la machine et la liberté d'action que l'on souhaite accorder à l'utilisateur. Ce problème est déjà présent dans un système d'exploitation et il relève de la notion de gestion des interruptions.

Nous pouvons trouver un compromis raisonnable dans le fait de découper les tâches internes en tâches séquentielles minimales ininterruptibles et en tâches interruptibles.

Les interruptions consisteront en actions potentielles de l'utilisateur sur la tâche en cours afin de :

- interrompre le travail en cours,
- quitter définitivement le logiciel,
- interroger un objet de sortie,
- lancer une commande exploratoire ...

Il faut donc qu'existe dans le système de développement du logiciel, un mécanisme qui permette de " *demander la main au système* " sans arrêter ni bloquer le reste de l'interface, ceci pendant le déroulement d'une action répétitive et longue. Lorsque l'interface a la main, l'utilisateur peut alors interrompre, quitter, interroger...

Ce mécanisme est disponible dans les RAD visuels pédagogiques (Delphi,Visual Basic, Visual C#), nous verrons comment l'implanter. Terminons ce tour d'horizon, par le dernier concept de base d'une interface : sa capacité à absorber certains dysfonctionnements.

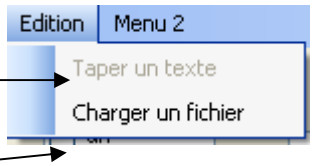
Concept

La résistance aux erreurs

Il faut en effet employer une méthode de programmation défensive afin de protéger le logiciel contre des erreurs comme par exemple des erreurs de manipulation de la part de l'utilisateur. Nous utilisons plusieurs outils qui concourent à la robustesse de notre logiciel. La protection est donc située à plusieurs niveaux.

1°) Une protection est apportée par le graphe événementiel qui n'autorise que certaines actions (activation-désactivation), matérialisé par un objet tel qu'un menu :

l'item " taper un fichier " n'est pas activable



l'item " Charger un fichier " est activable

2°) Une protection est apportée par le filtrage des données (on peut utiliser par exemple des logiciels d'automates de reconnaissance de la syntaxe des données).

3°) Un autre niveau de protection est apporté par les composants visuels utilisés qui sont sécurisés dans le RAD à l'origine. Par exemple la méthode LoadFile d'un richTextBox de C# qui permet le chargement d'un fichier dans un composant réagit d'une manière sécuritaire (c'est à dire rien ne se produit) lorsqu'on lui fournit un chemin erroné ou que le fichier n'existe pas.

4°) Un niveau de robustesse est apporté en C# par une utilisation des exceptions (semblable à C++ ou à Delphi) autorisant le détournement du code afin de traiter une situation interne anormale (dépassement de capacité d'un calcul, transtypage de données non conforme etc...). Le programmeur peut donc prévoir les incidents possibles et construire des gestionnaires d'exceptions.

Les événements avec

Plan général:

1. Construction de nouveaux événements

Design Pattern observer
Abonné à un événement
Déclaration d'un événement
Invocation d'un événement
Comment s'abonner à un événement
Restrictions et normalisation
Evénement normalisé avec informations
Evénement normalisé sans information

2. Les événements dans les Windows.Forms

Contrôles visuels et événements
Evénement Paint : avec information
Evénement Click : sans information
Code C# généré

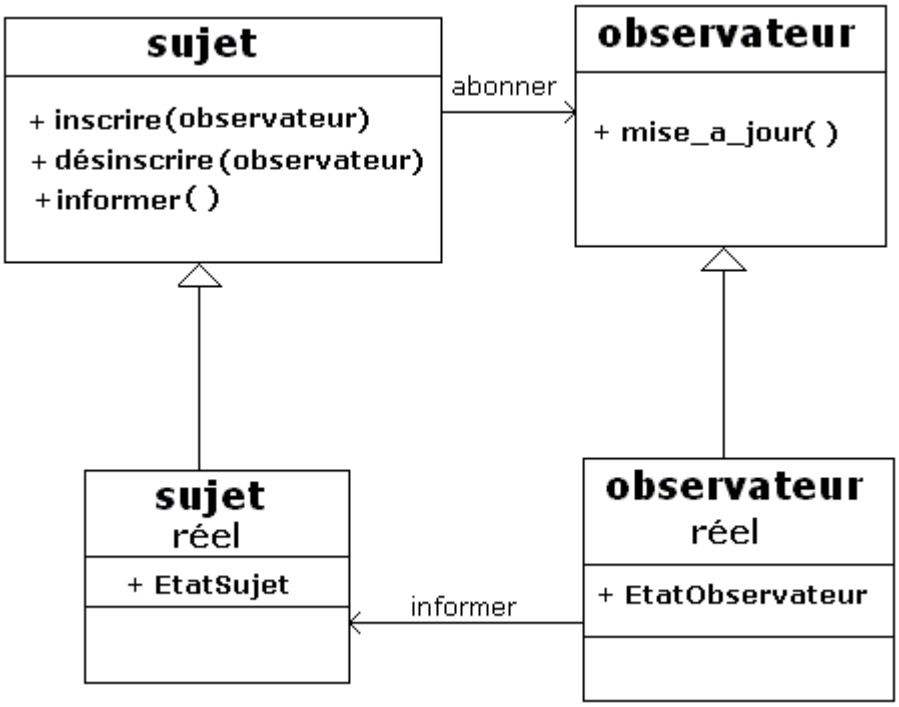
Rappel

Programmation orientée événements

Un programme objet orienté événements est construit avec des objets possédant des propriétés telles que les interventions de l'utilisateur sur les objets du programme et la liaison dynamique du logiciel avec le système déclenchent l'exécution de routines associées. Le traitement en programmation événementielle, consiste à mettre en place un mécanisme d'inteception puis de gestion permettant d'informer un ou plusieurs objets de la survenue d'un événement particulier.

1. Construction de nouveaux événements

Le modèle de conception de l'**observateur** (Design Pattern observer) est utilisé par Java et C# pour gérer un événement. Selon ce modèle, un client s'inscrit sur une liste d'abonnés auprès d'un observateur qui le préviendra lorsqu'un événement aura eu lieu. Les clients délèguent ainsi l'interception d'un événement à une autre entité. Java utilise ce modèle sous forme d'objet écouteur.



Design Pattern observer

Dans l'univers des Design Pattern on utilise essentiellement le modèle observateur dans les cas suivants :

- Quand le changement d'un objet se répercute vers d'autres.
- Quand un objet doit prévenir d'autres objets sans pour autant les connaitre.

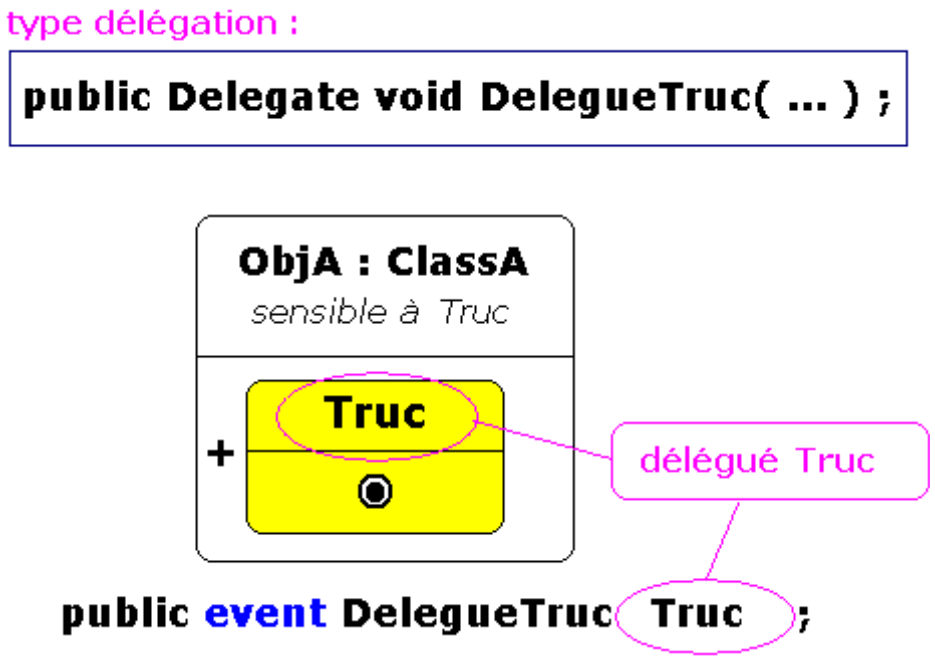
C# propose des mécanismes supportant les événements, mais avec une implémentation totalement différente de celle de Java. En observant le fonctionnement du langage nous pouvons dire que C# combine efficacement les fonctionnalités de Java et de Delphi.

Abonné à un événement

C# utilise les délégués pour fournir un mécanisme explicite permettant de gérer l'abonnement/notification.

En C# la délégation de l'écoute (gestion) d'un événement est confiée à un objet de type délégué : l'abonné est alors une méthode appelée gestionnaire de l'événement (contrairement à Java où l'abonné est une classe) acceptant les mêmes arguments et paramètres de retour que le délégué.

Déclaration d'un événement



Code C# :

```
using System;
using System.Collections;

namespace ExempleEvent {
   //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string )
   public delegate void DelegueTruc (string s);

   public class ClassA   {
      //--> déclaration d'une référence event de type délégué :
      public event DelegueTruc Truc;
   }
}
```

Invocation d'un événement

Une fois qu'une classe a déclaré un événement Truc, elle peut traiter cet événement exactement comme un délégué ordinaire. La démarche est très semblable à celle de Delphi, le champ Truc vaudra **null** si le client ObjA de ClassA n'a pas raccordé un délégué à l'événement Truc. En effet être sensible à plusieurs événements n'oblige pas chaque objet à gérer tous les événement, dans le cas où un objet ne veut pas gérer un événement Truc on n'abonne aucune méthode au délégué Truc qui prend alors la valeur **null**.

Dans l'éventualité où un objet doit gérer un événement auquel il est sensible, il doit invoquer l'événement Truc (qui référence une ou plusieurs méthodes). Invoquer un événement Truc consiste généralement à vérifier d'abord si le champ Truc est null, puis à appeler l'événement (le délégué Truc).

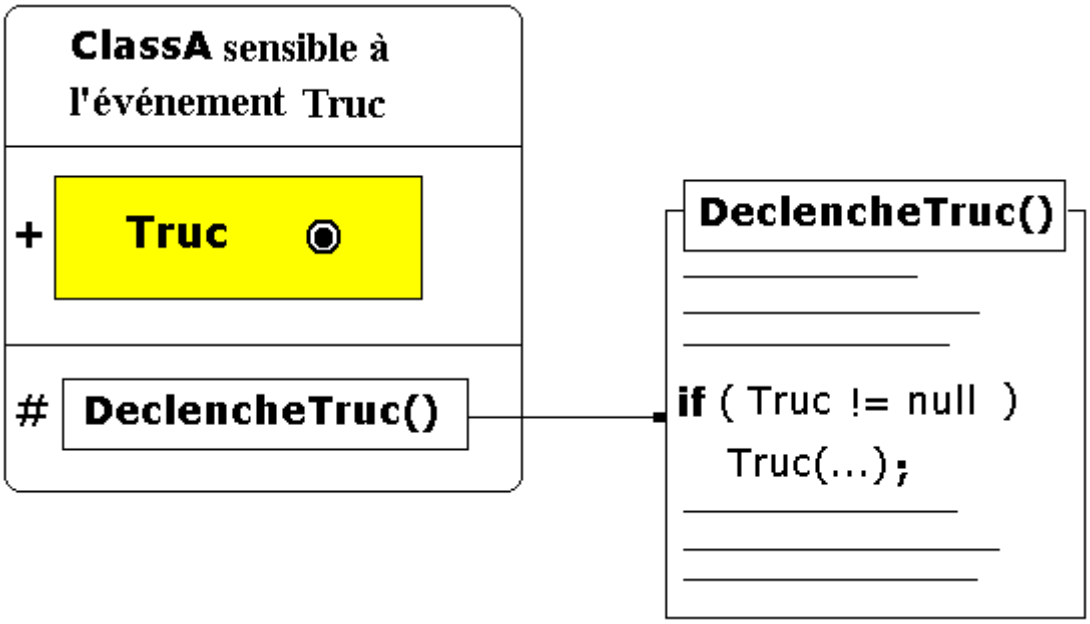
Remarque importante

L'appel d'un événement ne peut être effectué qu'à partir de la classe qui a déclaré cet événement.

Exemple construit pas à pas

Considérons ci-dessous la classe ClassA qui est sensible à un événement que nous nommons Truc (on déclare la référence Truc), dans le corps de la méthode **void** DeclencheTruc( ) on appelle l'événement Truc.

Nous déclarons cette méthode **void** DeclencheTruc( ) comme **virtuelle** et **protégée**, de telle manière qu'elle puisse être redéfinie dans la suite de la hiérarchie; ce qui constitue un gage d'évolutivité des futures classes quant à leur comportement relativement à l'événement Truc :



Il nous faut aussi prévoir une méthode **publique** qui permettra d'invoquer l'événement depuis une autre classe, nous la nommons LancerTruc.

Code C# , construisons progressivement notre exemple, voici les premières lignes du code :

```
using System;
using System.Collections;

namespace ExempleEvent {
   //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
   public delegate void DelegueTruc (string s);

   public class ClassA    {
      //--> déclaration d'une référence event de type délégué :
      public event DelegueTruc Truc;
      protected virtual void DeclencheTruc( ) {
         ....
         if ( Truc != null ) Truc("événement déclenché");
         ....
      }
      public void LancerTruc( ) {
         ....
         DeclencheTruc( ) ;
         ....
      }
   }
}
```

Comment s'abonner (se raccorder, s'inscrire, ...) à un événement

Un événement ressemble à un champ public de la classe qui l'a déclaré. Toutefois l'utilisation de ce champ est très restrictive, c'est pour cela qu'il est déclaré avec le spécificateur **event**. Seulement deux opérations sont possibles sur un champ d'événement qui rapellons-le est un délégué :

- Ajouter une nouvelle méthode (à la liste des méthodes abonnées à l'événement).
- Supprimer une méthode de la liste (désabonner une méthode de l'événement).

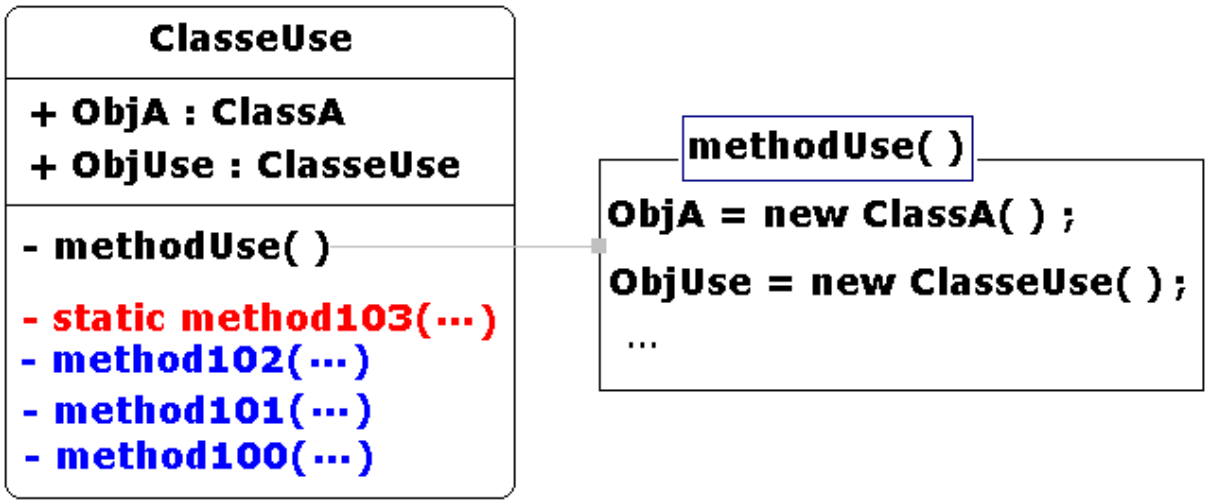
Enrichissons le code C# précédent avec la classe ClasseUse :

```
using System;
using System.Collections;

namespace ExempleEvent {
   //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
   public delegate void DelegueTruc (string s);

   public class ClassA    {
      //--> déclaration d'une référence event de type délégué :
      public event DelegueTruc Truc;

      protected virtual void DeclencheTruc( ) {
         ....
         if ( Truc != null ) Truc("événement déclenché");
         ....
      }
      public void LancerTruc( ) {
         ....
         DeclencheTruc( ) ;
         ....
      }
   }
   public class ClasseUse {
```

```
        static private void methodUse( ) {
            ClassA   ObjA = new ClassA( );
            ClasseUse ObjUse = new ClasseUse ( );
            //....
        }
        static public void Main(string[] x) {
            methodUse( ) ;
            //....
        }
    }
}
```

Il faut maintenant définir des gestionnaires de l'événement Truc (des méthodes ayant la même signature que le type délégation " **public delegate void** DelegueTruc (string s); ". Ensuite nous ajouterons ces méthodes au délégué Truc (nous les abonnerons à l'événement Truc), ces méthodes peuvent être de classe ou d'instance.

Supposons que nous ayons une méthode de classe et trois méthodes d'instances qui vont s'inscrire sur la liste des abonnés à Truc, ce sont quatre gestionnaires de l'événement Truc :

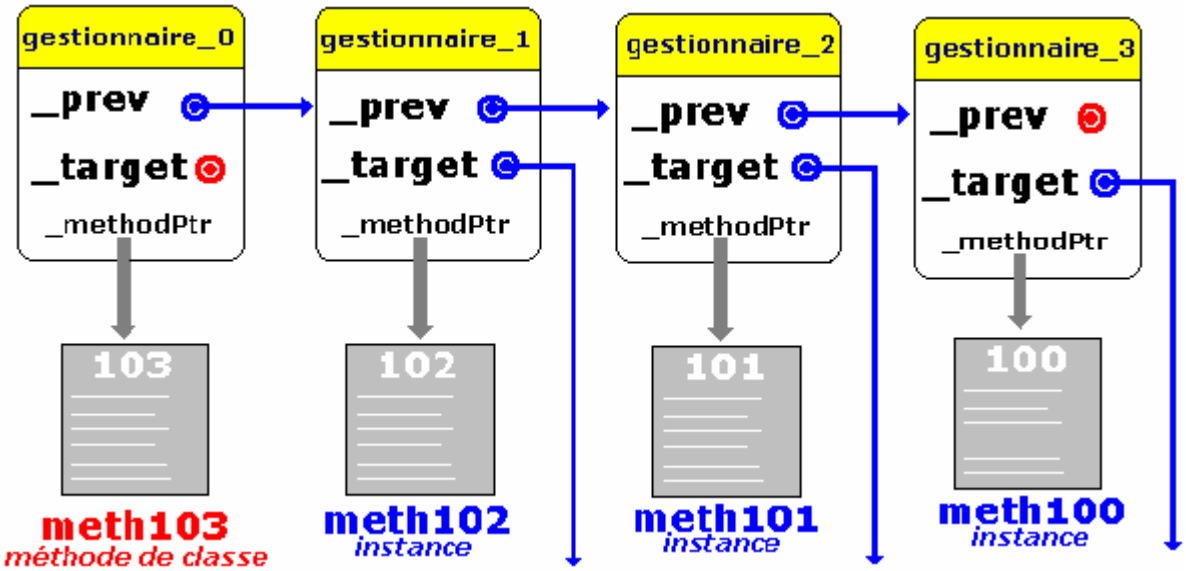


Ajoutons au code C# de la classe ClassA les quatre gestionnaires :

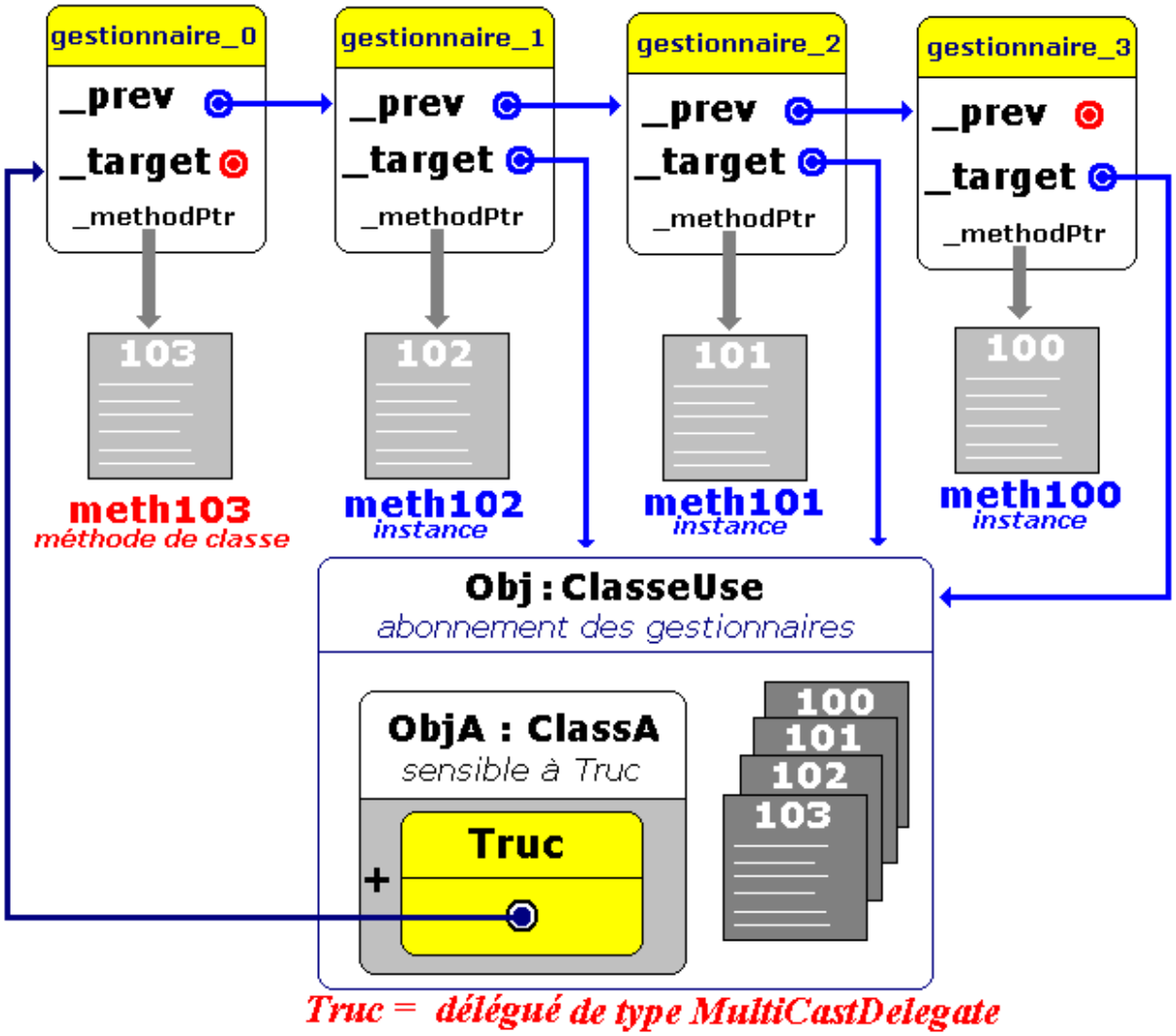
```
using System;
using System.Collections;

namespace ExempleEvent {
   //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
   public delegate void DelegueTruc (string s);

   public class ClassA   {
      //--> déclaration d'une référence event de type délégué :
      public event DelegueTruc Truc;

      protected virtual void DéclencheTruc( ) {
         ....
         if ( Truc != null ) Truc("événement déclenché")
         ....
      }
      public void LancerTruc( ) {
         ....
         DeclencheTruc( ) ;
```

```
            ....
        }
    }
    public class ClasseUse {
        public void method100(string s); {
            //...gestionnaire d'événement Truc: méthode d'instance.
        }
        public void method101(string s); {
            //...gestionnaire d'événement Truc: méthode d'instance.
        }
        public void method102(string s); {
            //...gestionnaire d'événement Truc: méthode d'instance.
        }
        static public void method103(string s); {
            //...gestionnaire d'événement Truc: méthode de classe.
        }
        static private void methodUse( ) {
            ClassA   ObjA = new ClassA( );
            ClasseUse ObjUse = new ClasseUse ( );
            //... il reste à abonner les  gestionnaires de l'événement Truc
        }
        static public void Main(string[ ] x) {
            methodUse( ) ;
        }
    }
}
```

Lorsque nous ajoutons en C# les nouvelles méthodes method100, ... , method103 au délégué Truc, par surcharge de l'opérateur +, nous dirons que les gestionnaires method100,...,method103, s'abonnent à l'événement Truc.



Prévenir (informer) un abonné correspond ici à l'action d'appeler l'abonné (appeler la méthode) :



Terminons le code C# associé à la figure précédente :

Nous complétons le corps de la méthode " **static private void** methodUse( ) " par l'abonnement au délégué des quatre gestionnaires.

Pour invoquer l'événement Truc, il faut pouvoir appeler enfin à titre d'exemple une invocation de l'événement :

```
using System;
using System.Collections;

namespace ExempleEvent {
   //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
   public delegate void DelegueTruc (string s);

   public class ClassA    {
      //--> déclaration d'une référence event de type délégué :
      public event DelegueTruc Truc;

      protected virtual void DeclencheTruc( ) {
         //....
```

```
            if ( Truc != null ) Truc("événement Truc déclenché");
            //....
        }
        public void LancerTruc( ) {
            //....
            DeclencheTruc( ) ;
            //....
        }
    }
    public class ClasseUse {
        public void method100(string s) {
            //...gestionnaire d'événement Truc: méthode d'instance.
            System.Console.WriteLine("information utilisateur : "+s);
        }
        public void method101(string s) {
            //...gestionnaire d'événement Truc: méthode d'instance.
            System.Console.WriteLine("information utilisateur : "+s);
        }
        public void method102(string s) {
            //...gestionnaire d'événement Truc: méthode d'instance.
            System.Console.WriteLine("information utilisateur : "+s);
        }
        static public void method103(string s) {
            //...gestionnaire d'événement Truc: méthode de classe.
            System.Console.WriteLine("information utilisateur : "+s);
        }
        static private void methodUse( ) {
            ClassA   ObjA = new ClassA( );
            ClasseUse ObjUse = new ClasseUse ( );
            //-- abonnement des gestionnaires:
            ObjA.Truc += new DelegueTruc (  ObjUse.method100 );
            ObjA.Truc += new DelegueTruc (  ObjUse.method101 );
            ObjA.Truc += new DelegueTruc (  ObjUse.method102 );
            ObjA.Truc += new DelegueTruc (  method103 );
            //-- invocation de l'événement:
            ObjA.LancerTruc( ) ;  //...l'appel à cette méthode permet d'invoquer l'événement Truc
        }
        static public void Main(string[] x) {
            methodUse( ) ;
        }
    }
}
```

Restrictions et normalisation .NET Framework

Bien que le langage C# autorise les événements à utiliser n'importe quel type délégué, le .NET Framework applique à ce jour à des fins de normalisation, certaines indications plus restrictives quant aux types délégués à utiliser pour les événements.

Les indications .NET Framework spécifient que le type délégué utilisé pour un événement doit disposer de deux paramètres et d'un retour définis comme suit :

- un paramètre de type **Object** qui désigne la source de l'événement,
- un autre paramètre soit de classe **EventArgs ,** soit d'une classe qui **dérive** de **EventArgs**, il encapsule toutes les informations personnelles relatives à l'événement,
- enfin le type du retour du délégué doit être **void**.

Evénement normalisé sans information :

Si vous n'utilisez pas d'informations personnelles pour l'événement, la signature du délégué sera :

public delegate void DelegueTruc (**Object** sender , **EventArgs** e) ;

Evénement normalisé avec informations :

Si vous utilisez des informations personnelles pour l'événement, vous définirez une classe **MonEventArgs** qui hérite de la classe **EventArgs** et qui contiendra ces informations personnelles, dans cette éventualité la signature du délégué sera :

public delegate void DelegueTruc (**Object** sender , **MonEventArgs** e) ;

Il est conseillé d'utiliser la représentation normalisée d'un événement comme les deux exemples ci-dessous le montre, afin d'augmenter la lisibilité et le portage source des programmes événementiels.

Mise en place d'un événement normalisé avec informations

Pour mettre en place un événement Truc normalisé contenant des informations personnalisées vous devrez utiliser les éléments suivants :

1°) une classe d'informations personnalisées sur l'événement
2°) une déclaration du type délégation normalisée (nom terminé par EventHandler)
*3°) une déclaration d'une référence Truc du type délégation normalisée spécifiée **event***
*4.1°) une méthode protégée qui déclenche l'événement Truc (nom commençant par **On**: OnTruc)*
4.2°) une méthode publique qui lance l'événement par appel de la méthode OnTruc
5°) un ou plusieurs gestionnaires de l'événement Truc
6°) abonner ces gestionnaires au délégué Truc
7°) consommer l'événement Truc

Exemple de code C# directement exécutable, associé à cette démarche :

```
using System;
using System.Collections;

namespace ExempleEvent {

//--> 1°) classe d'informations personnalisées sur l'événement

public class TrucEventArgs : EventArgs    {
    public string info ;
    public TrucEventArgs (string s) {
      info = s ;
    }
```

```
}
```

//--> 2°) déclaration du type délégation normalisé

```
public delegate void DelegueTrucEventHandler (  object sender,  TrucEventArgs e );
```

```
public class ClassA    {
```

//--> 3°) déclaration d'une référence event de type délégué :

```
    public event DelegueTrucEventHandler Truc;
```

//--> 4.1°) méthode protégée qui déclenche l'événement :

```
    protected virtual void OnTruc(  object sender,  TrucEventArgs e ) {
      //....
      if ( Truc != null ) Truc( sender , e );
      //....
    }
```

//--> 4.2°) méthode publique qui lance l'événement :

```
    public void LancerTruc( ) {
      //....
      TrucEventArgs evt = new TrucEventArgs ("événement déclenché" ) ;
      OnTruc ( this , evt );
      //....
    }

}

public class ClasseUse {
```

//--> 5°) les gestionnaires d'événement Truc

```
        public void method100(  object sender,  TrucEventArgs e ){
                //...gestionnaire d'événement Truc: méthode d'instance.
                System.Console.WriteLine("information utilisateur 100 : "+e.info);
        }

        public void method101(  object sender,  TrucEventArgs e ) {
                //...gestionnaire d'événement Truc: méthode d'instance.
                System.Console.WriteLine("information utilisateur 101 : "+e.info);
        }

        public void method102(  object sender,  TrucEventArgs e ) {
                //...gestionnaire d'événement Truc: méthode d'instance.
                System.Console.WriteLine("information utilisateur 102 : "+e.info);
        }

        static public void method103(  object sender,  TrucEventArgs e ) {
                //...gestionnaire d'événement Truc: méthode de classe.
                System.Console.WriteLine("information utilisateur 103 : "+e.info);
        }

static private void methodUse( ) {
        ClassA   ObjA = new ClassA( );
        ClasseUse ObjUse = new ClasseUse ( );
```

```
        //--> 6°) abonnement des gestionnaires:
         ObjA.Truc += new DelegueTrucEventHandler (  ObjUse.method100 );
         ObjA.Truc += new DelegueTrucEventHandler (  ObjUse.method101 );
         ObjA.Truc += new DelegueTrucEventHandler (  ObjUse.method102 );
         ObjA.Truc += new DelegueTrucEventHandler (  method103 );

        //--> 7°) consommation de l'événement:
         ObjA.LancerTruc( ) ;  //...l'appel à cette méthode permet d'invoquer l'événement Truc
      }
      static public void Main(string[] x) {
         methodUse( ) ;
      }
   }
}
```

Résultats d'exécution du programme console précédent :

```
D:\CsBuilder\EventPerso\bin\Debug\ProjEventPerso.exe
information utilisateur 100 : événement déclenché
information utilisateur 101 : événement déclenché
information utilisateur 102 : événement déclenché
information utilisateur 103 : événement déclenché
```

Mise en place d'un événement normalisé sans information

En fait, pour les événements qui n'utilisent pas d'informations supplémentaires personnalisées, le .NET Framework a déjà défini un type délégué approprié : **System.EventHandler** (équivalent au TNotifyEvent de Delphi):

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

Pour mettre en place un événement Truc normalisé sans information spéciale, vous devrez utiliser les éléments suivants :

> *1°) la classe System.EventArgs*
> *2°) le type délégation normalisée System.EventHandler*
> *3°) une déclaration d'une référence Truc du type délégation normalisée spécifiée **event***
> *4.1°) une méthode protégée qui déclenche l'événement Truc (nom commençant par **On**: OnTruc)*
> *4.2°) une méthode publique qui lance l'événement par appel de la méthode OnTruc*
> *5°) un ou plusieurs gestionnaires de l'événement Truc*
> *6°) abonner ces gestionnaires au délégué Truc*
> *7°) consommer l'événement Truc*

Remarque, en utilisant la déclaration **public delegate void** EventHandler (**object** sender, **EventArgs** e) contenue dans **System.EventHandler**, l'événement n'ayant aucune donnée personnalisée, le deuxième paramètre n'étant pas utilisé, il est possible de fournir le champ **static** Empty de la classe **EventArgs** .

EventArgs, membres

Constructeurs publics

EventArgs, constructeur

Initialise une nouvelle instance de la classe **EventArgs**.

Champs publics

S Empty

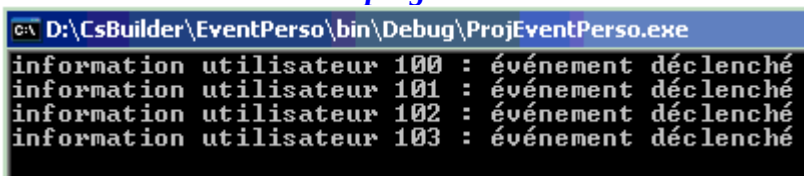
Représente un événement sans données d'événement.

Exemple de code C# directement exécutable, associé à un événement sans information personnalisée :

```
using System;
using System.Collections;

namespace ExempleEvent {

//--> 1°) classe d'informations personnalisées sur l'événement

System.EventArgs est déjà déclarée dans using System;

//--> 2°) déclaration du type délégation normalisé

System.EventHandler est déjà déclarée dans using System;

    public class ClassA   {
        //--> 3°) déclaration d'une référence event de type délégué :

        public event EventHandler Truc;

        //--> 4.1°) méthode protégée qui déclenche l'événement :

        protected virtual void OnTruc(  object sender,  EventArgs e ) {
          //....
          if ( Truc != null ) Truc( sender , e );
          //....
        }

        //--> 4.2°) méthode publique qui lance l'événement :

        public void LancerTruc( ) {
          //....
          OnTruc( this , EventArgs.Empty );
          //....
        }

    }

    public class ClasseUse {
          //--> 5°) les gestionnaires d'événement Truc

          public void method100(  object sender,  EventArgs e ){
```

```
            //...gestionnaire d'événement Truc: méthode d'instance.
            System.Console.WriteLine("information utilisateur 100 : événement déclenché");
        }

        public void method101(  object sender,  EventArgs e ) {
            //...gestionnaire d'événement Truc: méthode d'instance.
            System.Console.WriteLine("information utilisateur 101 : événement déclenché");
        }

        public void method102(  object sender,  EventArgs e ) {
            //...gestionnaire d'événement Truc: méthode d'instance.
            System.Console.WriteLine("information utilisateur 102 : événement déclenché");
        }

        static public void method103(  object sender,  EventArgs e ) {
            //...gestionnaire d'événement Truc: méthode de classe.
            System.Console.WriteLine("information utilisateur 103 : événement déclenché");
        }

        static private void methodUse( ) {
            ClassA   ObjA = new ClassA( );
            ClasseUse ObjUse = new ClasseUse ( );
            //--> 6°) abonnement des gestionnaires:
            ObjA.Truc += new DelegueTruc (  ObjUse.method100 );
            ObjA.Truc += new DelegueTruc (  ObjUse.method101 );
            ObjA.Truc += new DelegueTruc (  ObjUse.method102 );
            ObjA.Truc += new DelegueTruc (  method103 );

            //--> 7°) consommation de l'événement:
            ObjA.LancerTruc( ) ;  //...l'appel à cette méthode permet d'invoquer l'événement Truc
        }
        static public void Main(string[] x) {
            methodUse( ) ;
        }
    }
}
```

Résultats d'exécution de ce programme :



2. Les événements dans les Windows.Forms

Le code et les copies d'écran sont effectuées avec C#Builder 1.0 de Borland version personnelle gratuite.

Contrôles visuels et événements

L'architecture de fenêtres de .Net FrameWork se trouve essentiellement dans l'espace de noms

System.Windows.Forms qui contient des classes permettant de créer des applications contenant des IHM (interface humain machine) et en particulier d'utiliser les fonctionnalités afférentes aux IHM de Windows.

Plus spécifiquement, la classe System.Windows.Forms.Control est la classe mère de tous les composants visuels. Par exemple, les classes **Form**, **Button**, **TextBox**, etc... sont des descendants de la classe Control qui met à disposition du développeur C# 58 événements auxquels un contrôle est sensible.

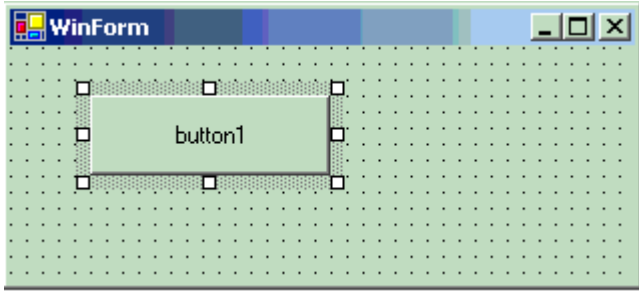
Ces 58 événements sont tous normalisés, certains sont des événements sans information spécifique, d'autres possèdent des informations spécifiques, ci-dessous un extrait de la liste des événements de la classe Control, plus particulièrement les événements traitant des actions de souris :

Control : Événements publics

BackColorChanged	Se produit lorsque la valeur de la propriété BackColor change.
BackgroundImageChanged	Se produit lorsque la valeur de la propriété BackgroundImage change.
.....	.....
Click	Se produit suite à un clic sur le contrôle.
.....	.....
MouseDown	Se produit lorsque le pointeur de la souris se trouve sur le contrôle et qu'un bouton de la souris est enfoncé.
MouseEnter	Se produit lorsque le pointeur de la souris se place dans le contrôle.
MouseHover	Se produit lorsque la souris pointe sur le contrôle.
MouseLeave	Se produit lorsque le pointeur de la souris s'écarte du contrôle.
MouseMove	Se produit lorsque le pointeur de la souris est placé sur le contrôle.
MouseUp	Se produit lorsque le pointeur de la souris se trouve sur le contrôle et qu'un bouton de la souris est relâché.
MouseWheel	Se produit lorsque la roulette de la souris bouge alors que le contrôle a le focus.
Move	Se produit lorsque le contrôle est déplacé.
Paint	Se produit lorsque le contrôle est redessiné.
.....	.....

Nous avons mis en évidence deux événements Click et Paint dont l'un est sans information (Click), l'autre est avec information (Paint). Afin de voir comment nous en servir nous traitons un exemple :

Soit une fiche (classe Form1 héritant de la classe Form) sur laquelle est déposé un bouton poussoir (classe Button) de nom **button1** :



Montrons comment nous programmons la réaction du bouton à un click de souris et à son redessinement. Nous devons faire réagir button1 qui est sensible à au moins 58 événements, aux deux événements Click et Paint. Rappelons la liste méthodologique ayant trait au cycle événementiel, on doit utiliser :

1°) une classe d'informations personnalisées sur l'événement
2°) une déclaration du type délégation normalisée (nom terminé par EventHandler)
3°) une déclaration d'une référence Truc du type délégation normalisée spécifiée ***event***
4.1°) une méthode protégée qui déclenche l'événement Truc (nom commençant par ***On****: OnTruc)*
4.2°) une méthode publique qui lance l'événement par appel de la méthode OnTruc
5°) un ou plusieurs gestionnaires de l'événement Truc
6°) abonner ces gestionnaires au délégué Truc
7°) consommer l'événement Truc

Les étapes 1° à 4° ont été conçues et developpées par les équipes de .Net et ne sont plus à notre charge.

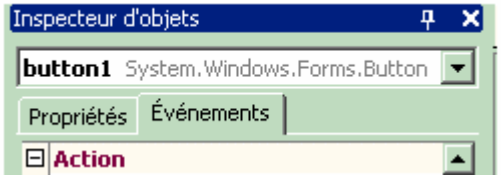
Il nous reste les étapes suivantes :

- 5°) à construire un gestionnaire de réaction de **button1** à l'événement Click et un gestionnaire de réaction de **button1**
- 6°) à abonner chaque gestionnaire au délégué correspondant (Click ou Paint)

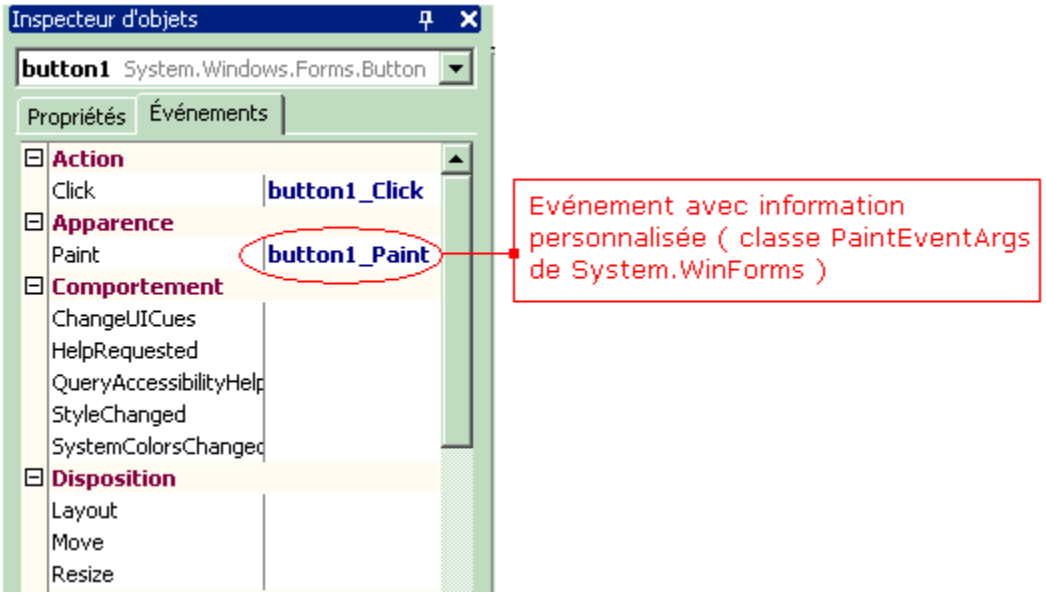
L'étape 7° est assurée par le système d'exploitation qui se charge d'envoyer des messages et de lancer les événements.

Evénement Paint : normalisé avec informations

Voici dans l'inspecteur d'objets de C#Builder l'onglet Evénements qui permet de visualiser le délégué à utiliser ainsi que le gestionnaire à abonner à ce délégué.



Dans le cas de l'événement Paint, le délégué est du type **PaintEventArgs** situé dans System.WinForms :



signature du délégué Paint dans System.Windows.Forms :

public delegate void PaintEventHandler (**object** sender, **PaintEventArgs** e);

La classe **PaintEventArgs** :

PaintEventArgs

Constructeur public

PaintEventArgs, constructeur

Initialise une nouvelle instance de la classe **PaintEventArgs** avec les graphiques et le rectangle de découpage spécifiés.

Propriétés publiques

ClipRectangle

Obtient le rectangle dans lequel peindre.

Graphics

Obtient le graphique utilisé pour peindre.

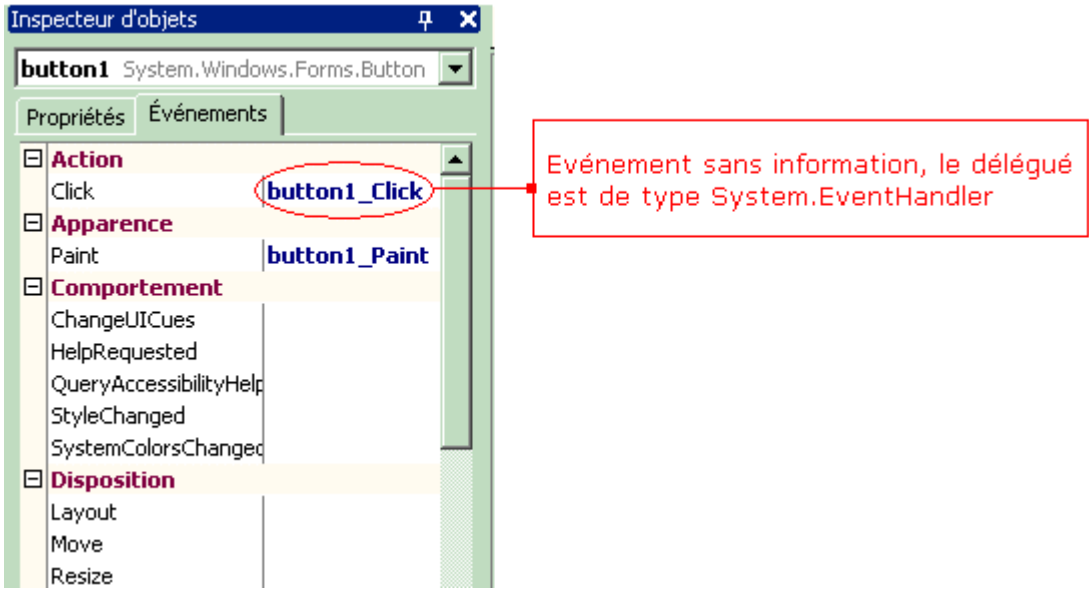
Méthodes publiques

Dispose

Surchargé. Libère les ressources utilisées par **PaintEventArgs**.

Evénement Click normalisé sans information

Dans le cas de l'événement Click, le délégué est de type **Event Handler** situé dans System :



signature du délégué Click dans System :

public delegate void EventHandler (**object** sender, **EventArgs** e);

En fait l'inspecteur d'objet de C#Builder permet de réaliser en mode visuel la machinerie des étapes qui sont à notre charge :

- le délégué de l'événement, [**public event** EventHandler Click;]
- le squelette du gestionnaire de l'événement, [**private void** button1_Click(object sender, System.EventArgs e){ }]
- l'abonnement de ce gestionnaire au délégué. [**this.**button1.Click += **new** System.EventHandler (**this.**button1_Click);]

Code C# généré

Voici le code généré par C#Builder utilisé en conception visuelle pour faire réagir button1 aux deux événements Click et Paint :

```
public class WinForm : System.Windows.Forms.Form    {
      private System.ComponentModel.Container components = null ;
      private System.Windows.Forms.Button button1;

      public WinForm( )  {
         InitializeComponent( );
      }

      protected override void Dispose (bool disposing) {
         if (disposing) {
            if (components != null )  {
               components.Dispose( ) ;
            }
         }
         base.Dispose(disposing) ;
      }
      private void InitializeComponent( ) {
         this.button1 = new System.Windows.Forms.Button( );
         ......

         this.button1.Click += new System.EventHandler( this.button1_Click );


         this.button1.Paint += new System.Windows.Forms.PaintEventHandler( this.button1_Paint );

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

      private void button1_Click(object sender, System.EventArgs e)       {
         //..... gestionnaire de l'événement OnClick
      }


      private void button1_Paint(object sender, System.Windows.Forms.PaintEventArgs e)       {
         //..... gestionnaire de l'événement OnPaint
      }

}
```

La machinerie événementielle est automatiquement générée par l'environnement RAD, ce qui épargne de nombreuses lignes de code au développeur et le laisse libre de penser au code spécifique de réaction à l'événement.

Propriétés et indexeurs en

Plan général:

1. Les propriétés

1.1 Définition et déclaration de propriété
1.2 Accesseurs de propriété
1.3 Détail et exemple de fonctionnement d'une propriété
Exemple du fonctionnement
Explication des actions
1.4 Les propriétés sont de classes ou d'instances
1.5 Les propriétés peuvent être masquées comme les méthodes
1.6 Les propriétés peuvent être virtuelles et redéfinies comme les méthodes
1.7 Les propriétés peuvent être abstraites comme les méthodes
1.8 Les propriétés peuvent être déclarées dans une interface
1.9 Exemple complet exécutable
1.9.1 Détail du fonctionnement en écriture
1.9.2 Détail du fonctionnement en lecture

2. Les indexeurs

2.1 Définitions et comparaisons avec les propriétés
2.1.1 Déclaration
2.1.2 Utilisation
2.1.3 Paramètres
2.1.4 Liaison dynamique abstraction et interface
2.2 Code C# complet compilable

1. Les propriétés

Les propriétés du langage C# sont très proches de celle du langage Delphi, mais elles sont plus complètes et restent cohérentes avec la notion de membre en C#.

1.1 Définition et déclaration de propriété

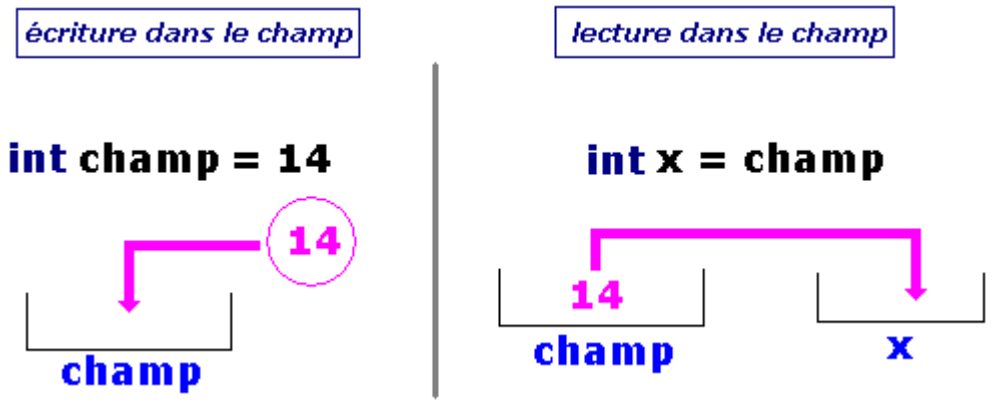
Définition d'une propriété

Une propriété définie dans une classe permet d'accéder à certaines informations contenues dans les objets instanciés à partir de cette classe. Une propriété a la même syntaxe de définition et d'utilisation que celle d'un champ d'objet (elle possède un type de déclaration), mais en fait elle invoque une ou deux méthodes internes pour fonctionner. Les méthodes internes sont déclarées à l'intérieur d'un bloc de définition de la propriété.

Déclaration d'une propriété propr1 de type **int** :

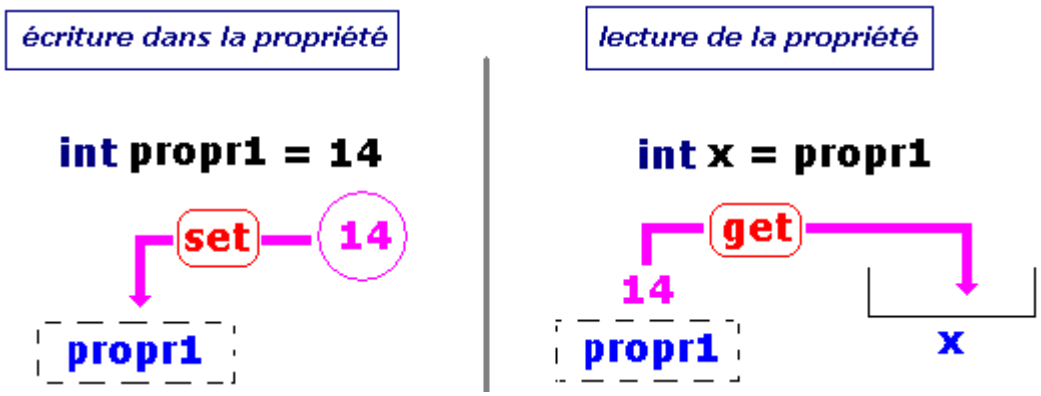
```
public int propr1 {
        //..... bloc de définition
}
```

Un champ n'est qu'un emplacement de stockage dont le contenu peut être consulté (***lecture du contenu du champ***) et modifié (***écriture dans le champ***), tandis qu'une propriété associe des **actions spécifiques à la lecture ou à l'écriture** ainsi que la modification des données que la propriété représente.



1.2 Accesseurs de propriété

En C#, une propriété fait systématiquement appel à une ou à deux méthodes internes dont les noms sont les mêmes pour toutes les propriétés afin de fonctionner soit en **lecture,** soit en **écriture.** On appelle ces méthodes internes des **accesseurs;** leur noms sont **get** et **set** , ci-dessous un exemple de lecture et d'écriture d'une propriété au moyen d'affectations :



Accesseur de lecture de la propriété :

Syntaxe : **get** { **return** ..... ;}

cet accesseur indique que la propriété est en lecture et doit renvoyer un résultat dont le type doit être le même que celui de la propriété. La propriété propr1 ci-dessous est déclarée en lecture seule et renvoie le contenu d'un champ de même type qu'elle :

```
private int champ;
public int propr1{
        get { return champ ; }
}
```

Accesseur d'écriture dans la propriété :

Syntaxe : **set** { ....}

cet accesseur indique que la propriété est en écriture et sert à initialiser ou à modifier la propriété. La propriété propr1 ci-dessous est déclarée en écriture seule et stocke une donnée de même type qu'elle dans la variable champ :

```
private int champ;
public int propr1{
        set { champ = value  ; }
}
```

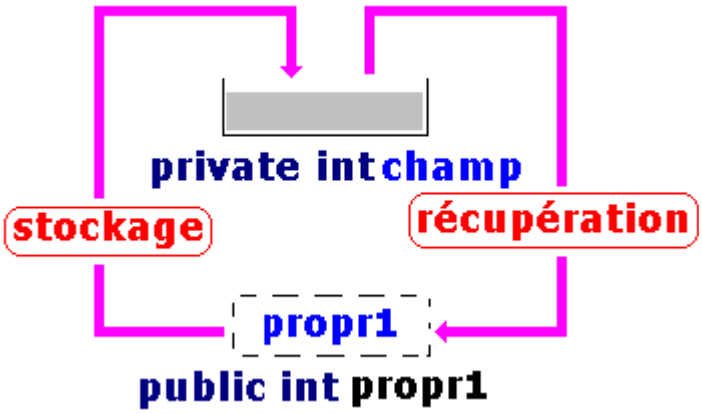
Le mot clef ***value*** est une sorte de paramètre implicite interne à l'accesseur **set**, il contient la valeur effective qui est transmise à la propriété lors de l'accès en écriture.

D'un manière générale lorsqu'une propriété fonctionne à travers un attribut (du même type que la propriété), l'attribut contient la donnée brute à laquelle la propriété permet d'accéder.

Ci-dessous une déclaration d'une propriété en lecture et écriture avec attribut de stockage :

```
private int champ;

public int propr1{
        get { return champ ; }
        set { champ = value  ; }
}
```

Le mécanisme de fonctionnement est figuré ci-après :



Dans l'exemple précédent, la propriété accède directement sans modification à la donnée brute stockée dans le champ, mais il est tout à fait possible à une propriété d'accéder à cette donnée en en modifiant sa valeur avant stockage ou après récupération de sa valeur.

1.3 Détail et exemple de fonctionnement d'une propriété

L'exemple ci-dessous reprend la propriété propr1 en lecture et écriture du paragraphe précédent et montre comment elle peut modifier la valeur brute de la donnée stockée dans l'attribut " **int** champ " :

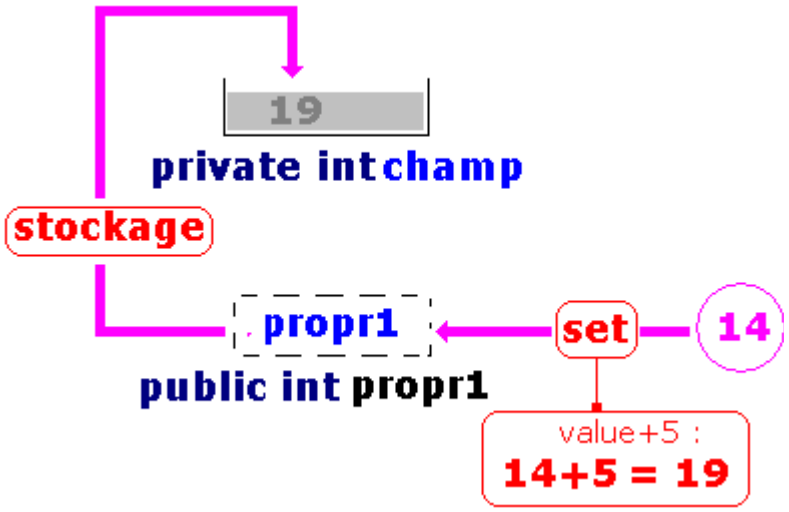
```
private int champ;

public int propr1{
        get {return champ*10;}
        set {champ = value + 5 ;}
}
```

Utilisons cette propriété en mode écriture à travers une affectation :

```
prop1 = 14 ;
```

Le mécanisme d'écriture est simulé ci-dessous :

*La valeur 14 est passée comme paramètre dans la méthode **set** à la variable implicite **value**, le calcul value+5 est effectué et le résultat 19 est stocké dans l'attribut champ.*

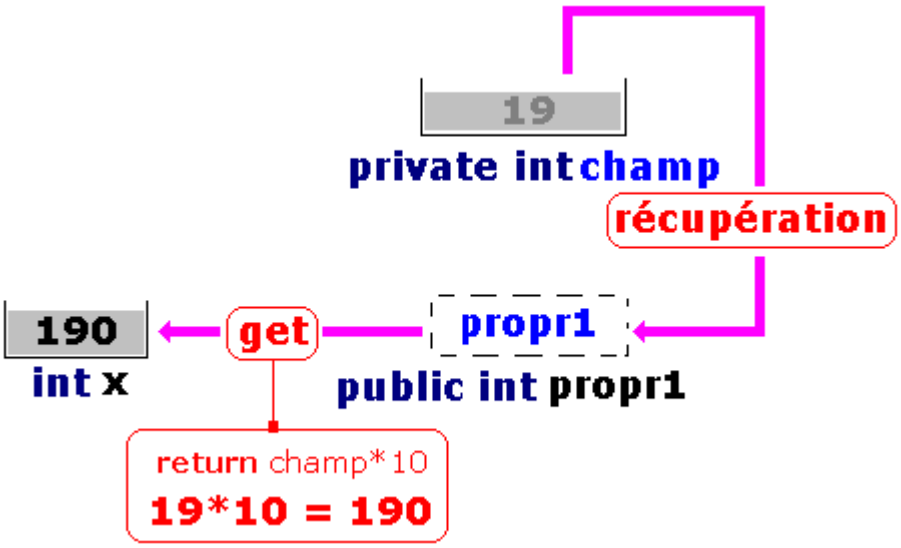


Utilisons maintenant notre propriété en mode lecture à travers une affectation :

```
int x = propr1 ;
```

Le mécanisme de lecture est simulé ci-dessous :

La valeur brute 19 stockée dans l'attribut champ est récupérée par la propriété qui l'utilise dans la méthode accesseur get en la multipliant par 10, c'est cette valeur modifiée de 190 qui renvoyée par la propriété.



Exemple pratique d'utilisation d'une propriété

Une propriété servant à fournir automatiquement le prix d'un article en y intégrant la TVA au taux de 19.6% et arrondi à l'unité d'euro supérieur :

```
private Double prixTotal ;
private Double tauxTVA = 1.196 ;

public Double prix {
        get {
                return Math.Round(prixTotal);
            }
        set {
                prixTotal = value * tauxTVA ;
            }
}
```

Ci-dessous le programme console C#Builder exécutable :

```
using System ;

namespace  ProjPropIndex
{
 class  Class  {
  static private  Double prixTotal  ;
  static private  Double tauxTVA  =  1.196  ;

  static public  Double prix   {
   get    {
    return  Math.Round ( prixTotal ) ;
   }
   set    {
    prixTotal  =  value  *  tauxTVA  ;
   }
  }

   [STAThread]
   static void  Main ( string [] args )  {
    Double val  =  100 ;
    System .Console.WriteLine ("Valeur entrée :" + val );
    prix  =  val ;
    System .Console.WriteLine ("Valeur stockée :" + prixTotal );
    val  =  prix ;
    System .Console.WriteLine ("valeur arrondie (lue) : " + val  ) ;
    System .Console.ReadLine ( );
  }
 }
}
```

Résultats d'exécution :

```
D:\CsharpExos\propIndex\bin\Debug\ProjProp
Valeur entrée :100
Valeur stockée :119,6
valeur arrondie (lue) : 120
```

Explications des actions exécutées :

On rentre 100€ dans la variable prix :

```
Double val = 100 ;
prix = val ;
```

Action effectuée :

On écrit 100 dans la propriété prix et celle-ci stocke 100*1.196=119.6 dans le champ prixTotal.

On exécute l'instruction :

```
val = prix ;
```

Action effectuée :

On lit la propriété qui arrondi le champ prixTotal à l'euro supérieur soit : 120€

1.4 Les propriétés sont de classes ou d'instances

Les propriétés, comme les champs peuvent être des **propriétés de classes** et donc qualifiées par les mots clefs comme **static**, **abstract** etc ...Dans l'exemple précédent nous avons qualifié tous les champs et la propriété prix en **static** afin qu'ils puissent être accessibles à la méthode **Main** qui est elle-même obligatoirement **static**.

Voici le même exemple utilisant une version avec des propriétés et des champs d'instances et non de classe (non static) :

```
using System ;

namespace ProjPropIndex
{
 class clA {
  private Double prixTotal ;
  private Double tauxTVA = 1.196 ;

  public Double prix {
   get { return Math.Round ( prixTotal ) ; }
   set { prixTotal = value * tauxTVA ; }
  }
 }

 class Class {

  [STAThread]
  static void Main ( string [] args ) {
    clA Obj = new clA ( );
    Double val = 100 ;
```

```
        System .Console.WriteLine ("Valeur entrée :" + val );
        Obj.prix  =  val ;
        // le champ prixTotal n'est pas accessible car il est privé
        val  =  Obj.prix ;
        System .Console.WriteLine ("valeur arrondie (lue) : " + val  ) ;
        System .Console.ReadLine ( );
    }
  }
}
```

Résultats d'exécution :

```
D:\CsharpExos\propIndex\bin\Debug\ProjPr
Valeur entrée :100
valeur arrondie (lue) : 120
```

1.5 Les propriétés peuvent être masquées comme les méthodes

Une propriété sans spécificateur particulier de type de liaison est considérée comme une entité à liaison statique par défaut.

Dans l'exemple ci-après nous dérivons une nouvelle classe de la classe clA nommée clB, nous redéclarons dans la classe fille une nouvelle propriété ayant le même nom, à l'instar d'un champ ou d'une méthode C# considère que nous masquons la propriété mère et nous suggère le conseil suivant :

[C# Avertissement] Class..... : Le mot clé new est requis sur '.....', car il masque le membre hérité..... '

Nous mettons donc le mot clef **new** devant la nouvelle déclaration de la propriété dans la classe fille. En reprenant l'exemple précédent supposons que dans la classe fille clB, la TVA soit à 5%, nous redéclarons dans clB une propriété prix qui va masquer celle de la mère :

```
using  System ;

namespace  ProjPropIndex
{
 class  clA  {
   private  Double prixTotal  ;
   private  Double tauxTVA  =  1.196  ;

   public  Double prix    { // propriété de la classe mère
     get    {  return  Math.Round ( prixTotal ) ;  }
     set    {  prixTotal  =  value  *  tauxTVA  ;  }
   }
 }
 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  {
```

```
    [STAThread]
   static void  Main ( string [] args )   {
    clA Obj  =  new  clA ( );
    Double val  =  100 ;
    System .Console.WriteLine ("Valeur entrée clA Obj :" + val );
    Obj.prix  =  val ;
    val  =  Obj.prix ;
    System .Console.WriteLine ("valeur arrondie (lue)clA Obj : " + val  ) ;
    System .Console.WriteLine ("-------------------------------------");
    clB Obj2  =  new  clB ( );
    val  =  100 ;
    System .Console.WriteLine ("Valeur entrée clB Obj2 :" + val );
    Obj2.prix  =  val ;
    val  =  Obj2.prix ;
    System .Console.WriteLine ("valeur arrondie (lue)clB Obj2: " + val  ) ;
    System .Console.ReadLine ( );
   }
  }
 }
```

Résultats d'exécution :

```
D:\CsharpExos\propIndex\bin\Debug\ProjProp
Valeur entrée clA Obj :100
valeur arrondie (lue)clA Obj : 120
---------------------------------
Valeur entrée clB Obj2 :100
valeur arrondie (lue)clB Obj2: 105
```

1.6 Les propriétés peuvent être virtuelles et redéfinies comme les méthodes

Les propriété en C# ont l'avantage important d'être utilisables dans le contexte de liaison dynamique d'une manière strictement identique à celle des méthodes en C# , ce qui confère au langage une "orthogonalité" solide relativement à la notion de polymorphisme.

Une propriété peut donc être déclarée virtuelle dans une classe de base et être surchargée dynamiquement dans les classes descendantes de cette classe de base.

Dans l'exemple ci-après semblable au précédent, nous déclarons dans la classe mère **clA** la propriété prix comme **virtual**, puis :

- Nous dérivons **clB**, une classe fille de la classe **clA** possédant une propriété prix masquant statiquement la propriété virtuelle de la classe **clA**, dans cette classe **clB** la TVA appliquée à la variable prix est à 5% (nous mettons donc le mot clef **new** devant la nouvelle déclaration de la propriété prix dans la classe fille **clB**). La propriété prix est dans cette classe **clB** à liaison statique.

- Nous dérivons une nouvelle classe de la classe **clA** nommée **clB2** dans laquelle nous redéfinissons en **override** la propriété prix ayant le même nom, dans cette classe **clB2** la TVA appliquée à la variable prix est aussi à 5%. La propriété prix est dans cette classe **clB2** à liaison dynamique.

Notre objectif est de comparer les résultats d'exécution obtenus lorsque l'on utilise une référence

d'objet de classe mère instanciée soit en objet de classe **clB** ou **clB2**. C'est le comportement de la propriété prix dans chacun de deux cas (statique ou dynamique) qui nous intéresse :

```
using System ;

namespace ProjPropIndex
{
 class clA {
  private Double prixTotal ;
  private Double tauxTVA = 1.196 ;

  public virtual Double prix { // propriété virtuelle de la classe mère
   get { return Math.Round ( prixTotal ) ; }
   set { prixTotal = value * tauxTVA ; }
  }
 }
 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 clB2 : clA {
  private Double prixLocal ;
  public override Double prix {// redéfinition de la propriété de la classe mère
   get { return Math.Round ( prixLocal ) ; }
   set { prixLocal = value * 1.05 ; }
  }
 }

 class Class {
  static private Double prixTotal ;
  static private Double tauxTVA = 1.196 ;

  static public Double prix {
   get { return Math.Round ( prixTotal ) ; }
   set { prixTotal = value * tauxTVA ; }
  }

  [STAThread]
  static void Main ( string [] args ) {
   clA Obj = new clA ( );
   Double val = 100 ;
   System.Console.WriteLine ("Valeur entrée Obj=new clA :" + val );
   Obj.prix = val ;
   val = Obj.prix ;
   System.Console.WriteLine ("valeur arrondie (lue)Obj=new clA : " + val ) ;
   System.Console.WriteLine ("----------------------------------------");
   Obj = new clB ( );
   val = 100 ;
   System.Console.WriteLine ("Valeur entrée Obj=new clB :" + val );
   Obj.prix = val ;
   val = Obj.prix ;
   System.Console.WriteLine ("valeur arrondie (lue)Obj=new clB : " + val ) ;
   System.Console.WriteLine ("----------------------------------------");
   Obj = new clB2 ( );
   val = 100 ;
   System.Console.WriteLine ("Valeur entrée Obj=new clB2 :" + val );
   Obj.prix = val ;
```

```
  val  =  Obj.prix ;
  System.Console.WriteLine ("valeur arrondie (lue)Obj=new clB2 : " + val  ) ;
  System.Console.ReadLine ( );
 }
}
}
```

Résultats d'exécution :

```
D:\CsharpExos\propIndex\bin\Debug\ProjPropIndex.exe
Valeur entrée  Obj=new clA :100
valeur arrondie (lue)Obj=new clA  : 120
----------------------------------------
Valeur entrée  Obj=new clB  :100
valeur arrondie (lue)Obj=new clB : 120
----------------------------------------
Valeur entrée  Obj=new clB2  :100
valeur arrondie (lue)Obj=new clB2 : 105
```

Nous voyons bien que le même objet Obj instancié en classe **clB** ou en classe **clB2** ne fournit pas les mêmes résultats pour la propriété prix, ces résulats sont conformes à la notion de polymorphisme en particulier pour l'instanciation en **clB2**.

Rappelons que le **masquage statique** doit être utilisé comme pour les méthodes à bon escient, plus spécifiquement lorsque nous ne souhaitons pas utiliser le polymorphisme, dans le cas contraire c'est la **liaison dynamique** qui doit être utilisée pour **définir et redéfinir des propriétés**.

1.7 Les propriétés peuvent être abstraites comme les méthodes

Les propriétés en C# peuvent être déclarées **abstract**, dans ce cas comme les méthodes elles sont automatiquement virtuelles sans necéssiter l'utilisation du mot clef **virtual**.

Comme une méthode abstraite, une propriété abstraite n'a pas de corps de définition pour le ou les accesseurs qui la composent, ces accesseurs sont implémentés dans une classe fille.

Toute classe déclarant une propriété **abstract** doit elle-même être déclarée **abstract**, l'implémentation de la propriété a lieu dans une classe fille, soit en masquant la propriété de la classe mère (grâce à une déclaration à liaison statique avec le mot clef **new**), soit en la redéfinissant (grâce à une déclaration à liaison dynamique avec le mot clef **override**) :

```
abstract class  clA  {
  public abstract Double prix    { // propriété abstraite virtuelle de la classe mère
    get   ;  // propriété abstraite en lecture
    set   ; // propriété abstraite en écriture
  }
 }
 class  clB1  :  clA  {
  private  Double prixTotal  ;
  private  Double tauxTVA  =  1.196  ;


  public new  Double prix    { //--redéfinition par new refusée (car membre abstract)
```

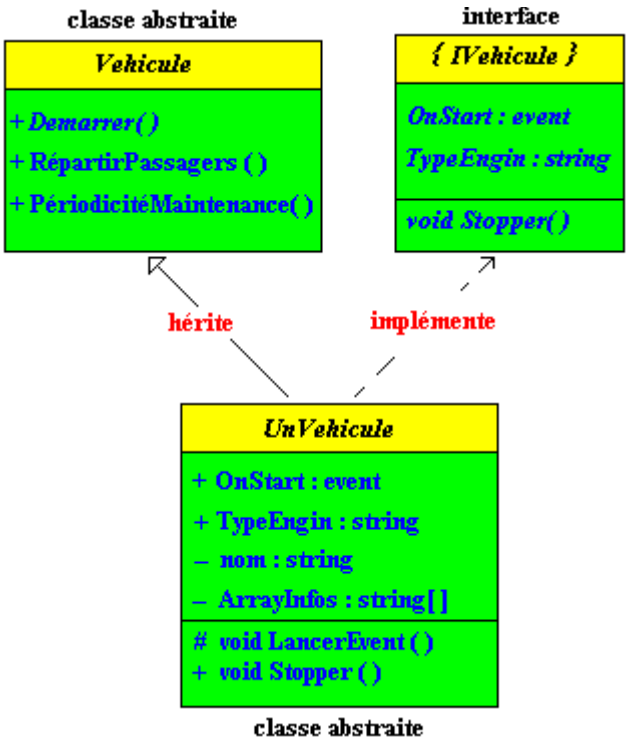
```
    get    { return  Math.Round (prixTotal ) ;  }
    set    {   prixTotal  =  value  *  tauxTVA  ;   }
  }
 }

class  clB2  :  clA   {
   private  Double prixTotal  ;
   private  Double tauxTVA  =  1.05  ;
  public override  Double prix    { //  redéfinition de la propriété par override correcte
    get    { return  Math.Round (prixTotal ) ;  }
    set    {   prixTotal  =  value  *  tauxTVA  ;   }
  }
 }
```

1.8 Les propriétés peuvent être déclarées dans une interface

Les propriétés en C# peuvent être déclarées dans une interface comme les événements et les méthodes sans le mot clef **abstract**, dans ce cas comme dans le cas de propriété abstraites la déclaration ne contient pas de corps de définition pour le ou les accesseurs qui la composent, ces accesseurs sont implémentés dans une classe fille qui implémente elle-même l'interface.

Les propriétés déclarées dans une interface lorsqu'elles sont implémentées dans une classe peuvent être définies soit à liaison statique, soit à liaison dynamique.

Ci dessous une exemple de hiérarchie abstraite de véhicules, avec une interface IVehicule contenant un événement (*cet exemple est spécifié au chapitre sur les interfaces*) :



```
abstract class  Vehicule  {  // classe abstraite mère
   ....
}
```

```
interface  IVehicule {
   ....
   string  TypeEngin    { // déclaration de propriété abstraite par défaut
      get ;
      set ;
   }
    ....
}
```

```
abstract class  UnVehicule : Vehicule , IVehicule  {
  private  string  nom = "";
  ....
  public virtual  string  TypeEngin  {    // implantation virtuelle de la propriété
     get      {        return  nom ;       }
     set      {         nom = "["+value+"]" ;       }
  }
  ....
}
```

```
abstract class  Terrestre  :  UnVehicule {
 ....
 public override  string  TypeEngin     { // redéfinition de propriété
  get  {  return  base .TypeEngin ;  }
  set  {  string  nomTerre = value + "-Terrestre";
            base .TypeEngin = nomTerre ;  }
 }
}
```

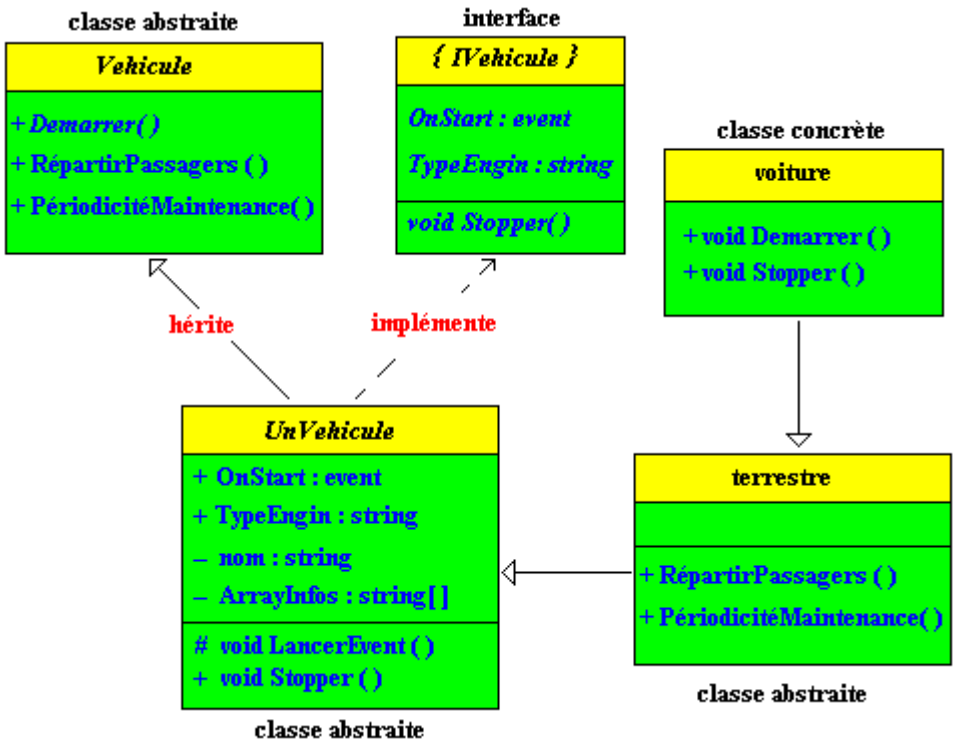
1.9 Exemple complet exécutable

Code C# complet compilable avec l'événement et une classe concrète

```
   public delegate void Starting ( );   // delegate declaration de type pour l'événement

   abstract class  Vehicule  {  // classe abstraite mère
     public abstract void  Demarrer ( ); // méthode abstraite
     public void  RépartirPassagers ( )  {  } // implantation de méthode avec corps vide
     public void  PériodicitéMaintenance ( ) {  } // implantation de méthode avec corps vide
   }
   interface  IVehicule {
      event  Starting OnStart ;   // déclaration d'événement du type délégué : Starting
      string  TypeEngin    { // déclaration de propriété abstraite par défaut
        get ;
        set ;
      }
       void  Stopper ( ); // déclaration de méthode abstraite par défaut
```

```
}

//-- classe abstraite héritant de la classe mère et implémentant l'interface :
abstract class  UnVehicule : Vehicule , IVehicule  {
  private  string  nom = "";
  private  string [ ] ArrayInfos = new  string [10] ;
  public  event  Starting OnStart ;
  protected void  LancerEvent ( )   {
    if( OnStart  !=  null)
       OnStart ( );
  }
  public virtual  string  TypeEngin  {    // implantation virtuelle de la propriété
     get      {        return  nom ;       }
     set      {         nom = "["+value+"]" ;       }
  }
  public virtual void  Stopper ( )   {  } // implantation virtuelle de méthode avec corps vide
}

abstract class  Terrestre  :  UnVehicule {
 private  string  nomTerre = "";
 public new void  RépartirPassagers ( ) {   }
 public new void  PériodicitéMaintenance ( ) {  }
 public override  string  TypeEngin     { // redéfinition de propriété
  get  {   return  base.TypeEngin ;  }
  set  {   string  nomTerre = value + "-Terrestre";
            base.TypeEngin = nomTerre ;  }
 }
}

class Voiture  : Terrestre   {
      public override  string  TypeEngin     { // redéfinition de propriété
        get  {   return  base.TypeEngin + "-voiture";  }
       set  {    base.TypeEngin =  "(" + value + ")";   }
      }
      public override void  Demarrer ( )  {
         LancerEvent( );
      }
      public override void  Stopper ( )  {
         //...
      }
}



class UseVoiture   { // instanciation d'une voiture particulière
      static void  Main ( string [] args )  {
         UnVehicule x =  new Voiture ( ) ;
         x.TypeEngin  = "Picasso" ; // propriété en écriture
         System.Console.WriteLine  ( "x est une " + x.TypeEngin  ) ; // propriété en lecture
         System.Console.ReadLine ( ) ;
      }
}
```

Diagrammes de classes UML de la hiérarchie implantée :



Résultats d'exécution :

```
D:\CsharpExos\propIndex\bin\Debug\ProjPropIndex.exe
x est une [(Picasso)-Terrestre]-voiture
```

1.9.1 Détails de fonctionnement de la propriété TypeEngin en écriture

La propriété TypeEngin est en écriture dans :

```
x.TypeEngin = "Picasso" ;
```

Elle remonte à ses définitions successives grâce l'utilisation du mot clef base qui fait référence à la classe mère de la classe en cours.

- **propriété TypeEngin dans la classe Voiture :**

  (Picasso)

  base.TypeEngin

- **propriété TypeEngin dans la classe Terrestre :**

  (Picasso)-Terrestre

  ..... **base.TypeEngin**

- **propriété TypeEngin dans la classe UnVehicule :**

  [(Picasso)-Terrestre]

  ..... **private string nom**

Définition de la propriété dans la classe Voiture (écriture) :

```
x.TypeEngin = "Picasso" ;

class Voiture : Terrestre { ...
public override string TypeEngin { // redéfinition de propriété
        ...
        set { base.TypeEngin = "(" + value + ")"; }
}
... }

// value = "Picasso" => base.TypeEngin = "(Picasso)" :
```

(Picasso)

base.TypeEngin

base **référence ici la classe** Terrestre**.**

Définition de la propriété dans la classe Terrestre (écriture) :

(Picasso)

value

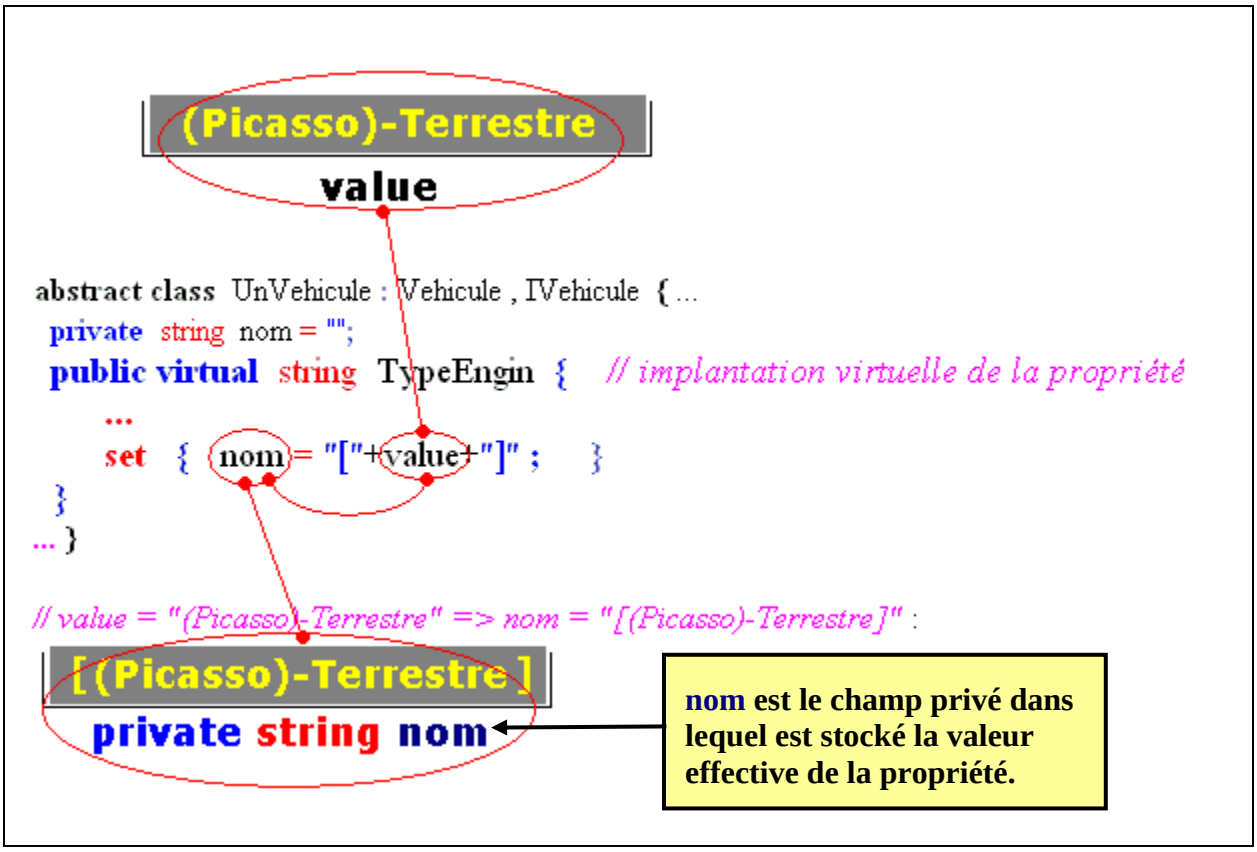
```
abstract class Terrestre : UnVehicule { ...
public override string TypeEngin { // redéfinition de propriété
  ...
  set { string nomTerre = value + "-Terrestre";
        base.TypeEngin = nomTerre ; }
}
... }

// value = "(Picasso)" => base.TypeEngin = "(Picasso)-Terrestre" :
```

(Picasso)-Terrestre

base.TypeEngin

base **référence ici la classe** UnVehicule**.**

Définition de la propriété dans la classe UnVehicule (écriture) :

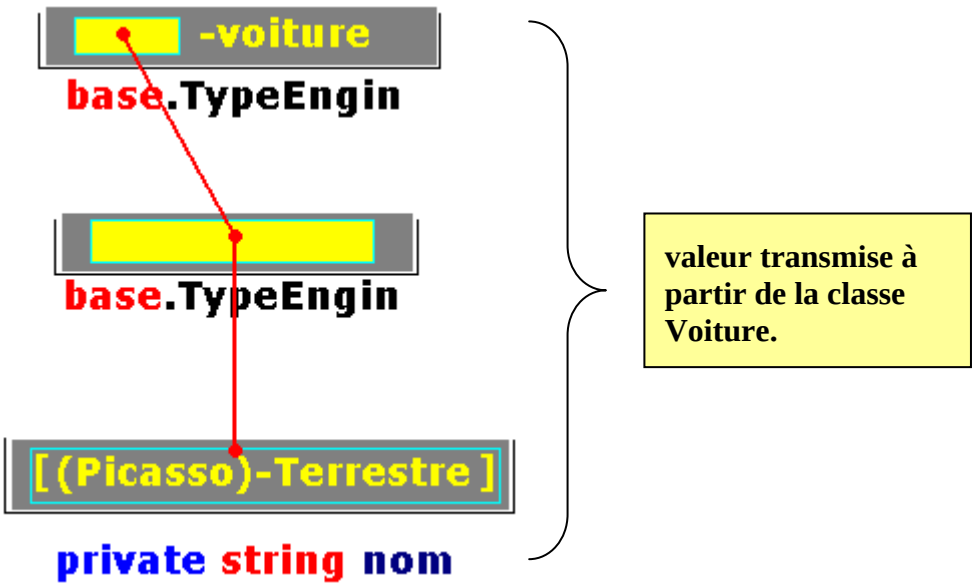


1.9.2 Détails de fonctionnement de la propriété TypeEngin en lecture

La propriété TypeEngin est en lecure dans :

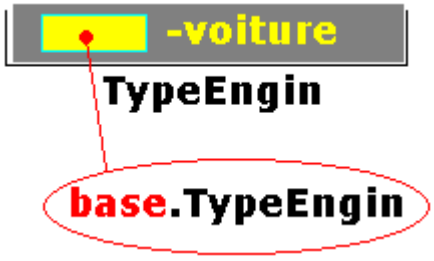
```
System.Console.WriteLine ( "x est une " + x.TypeEngin ) ;
```

Pour aller chercher la valeur effective, elle remonte à ses définitions successives grâce l'utilisation du mot clef base qui fait référence à la classe mère de la classe en cours.



Définition de la propriété dans la classe Voiture (lecture) :

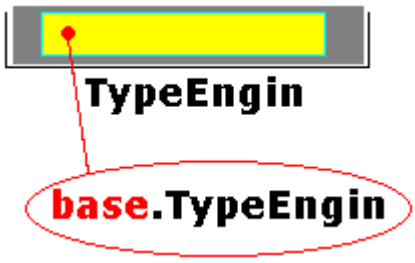
```
class Voiture  : Terrestre { ...
  public override  string  TypeEngin     { // redéfinition de propriété
        get  {  return  base.TypeEngin + "-voiture";  }
        ...
  }
... }
```



L'accesseur get va chercher le résultat dans base.TypeEngin et lui concatène le mot "-voiture".
base.TypeEngin référence ici la propriété dans la classe Terrestre.

Définition de la propriété dans la classe Terrestre (lecture) :

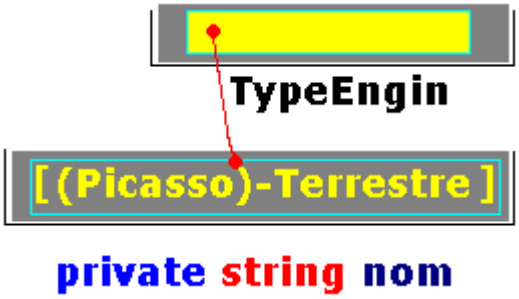
```
abstract class  Terrestre  :  UnVehicule { ...
  public override  string  TypeEngin     { // redéfinition de propriété
     get  {  return  base.TypeEngin ;  }
     ...
  }
... }
```



L'accesseur get va chercher le résultat dans base.TypeEngin.
base.TypeEngin référence ici la propriété dans la classe UnVehicule.

Définition de la propriété dans la classe UnVehicule (lecture) :

```
abstract class  UnVehicule : Vehicule , IVehicule  { ...
  private  string  nom = "";
  public virtual  string  TypeEngin  {    // implantation virtuelle de la propriété
     get  {  return  nom ;  }
     ...
  }
... }
```



L'accesseur get va chercher le résultat dans le champ nom.
nom est le champ privé dans lequel est stocké la valeur effective de la propriété.

1.10 les accesseurs get et set peuvent avoir une visibilité différente

Depuis la version 2.0, un propriété peut être "public", son accesseur d'écriture **set** (ou de lecture **get**) peut être par exemple qualifié de "**protected**" afin de ne permettre l'écriture (ou la lecture) que dans des classes filles.

Les modificateurs de visibilité des accesseurs get et set doivent être, lorsqu'ils sont présents plus restrictifs que ceux de la propriété elle-même.

Exemple de propriété sans modificateurs dans get et set :

```
class clA    {
        private int champ;
        public int prop
        {
            get { return champ; }
            set { champ = value; }
        }
    }
```

Exemple de propriété avec modificateurs dans **get** et **set** :

```
class clA    {
        private int champ;
        public int prop
        {
            protected get { return champ; } //lecture dans classe fille
            protected set { champ = value; }//écriture dans classe fille
        }
    }
```

2. Les indexeurs

Nous savons en Delphi qu'il existe une notion de propriété par défaut qui nous permet par exemple dans un objet Obj de type TStringList se nomme **strings**, d'écrire **Obj[5]** au lieu de **Obj.strings[5]**. La notion d'indexeur de C# est voisine de cette notion de propriété par défaut en Delphi.

2.1 Définitions et comparaisons avec les propriétés

Un indexeur est un membre de classe qui permet à un objet d'être **indexé de la même manière qu'un tableau.** La signature d'un indexeur doit être différente des signatures de tous les autres indexeurs déclarés dans la même classe. Les indexeurs et les propriétés sont très similaires de par leur concept, c'est pourquoi nous allons définir les indexeurs à partir des propriétés.

Tous les indexeurs sont représentés par l' opérateur **[ ]** . Les liens sur les propriétés ou les indexeurs du tableau ci-dessous renvoient directement au paragraphe associé.

Propriété	Indexeur
Déclarée par son nom.	Déclaré par le mot clef **this**.
Identifiée et utilisée par son nom.	Identifié par sa signature, utilisé par l'opérateur **[ ]** . L'opérateur **[ ]** doit être situé immédiatement après le nom de l'objet.
Peut être un membre de classe **static** ou un membre d'instance.	**Ne** peut **pas** être un membre **static**, est toujours un membre d'instance.
L'accesseur **get** correspond à une méthode sans paramètre.	L'accesseur **get** correspond à une méthode pourvue de la même liste de paramètres formels que l'indexeur.
L'accesseur **set** correspond à une méthode avec un seul paramètre implicite **value**.	L'accesseur **set** correspond à une méthode pourvue de la même liste de paramètres formels que l'indexeur plus le paramètre implicite **value**.
Une propriété Prop héritée est accessible par la syntaxe base.**Prop**	Un indexeur Prop hérité est accessible par la syntaxe base.**[ ]**
Les propriétés peuvent être à liaison statique, à liaison dynamique, masquées ou redéfinies.	Les indexeurs peuvent être à liaison statique, à liaison dynamique, masqués ou redéfinis.

Les propriétés peuvent être abstraites.	Les indexeurs peuvent être abstraits.
Les propriétés peuvent être déclarées dans une interface.	Les indexeurs peuvent être déclarés dans une interface.

2.1.1 Déclaration

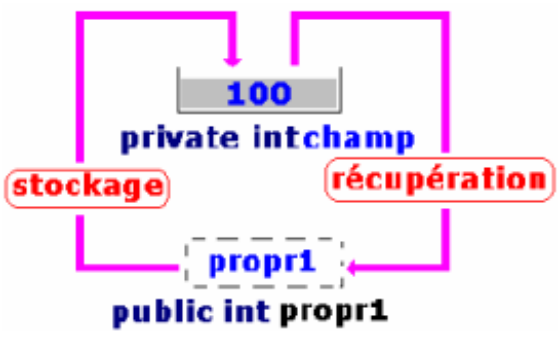
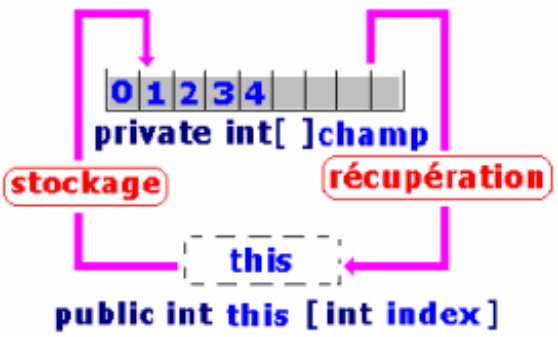
Propriété	Indexeur

Déclarée par son nom, avec champ de stockage :
private int champ;
 	
public int propr1**{**
  get { return champ ; **}**
  set { champ = ***value*** ; **}**
 	
}
 	
private intchamp
stockage récupération
propr1
public int propr1
	Déclaré par le mot clef **this**, avec champ de stockage :
	private int [ ] champ = **new int** [10];
	
	public int this [**int** index]**{**
	  get { return champ[index] ; **}**
	  set { champ[index] = ***value*** ; **}**
	}
	
	private int[ ]champ
	stockage récupération
	this
	public int this [int index]

2.1.2 Utilisation

Propriété	Indexeur

Déclaration :
class clA **{**
  private int champ;
 	
  public int propr1**{**
  get { return champ ; **}**
  set { champ = ***value*** ; **}**
  }
}
Utilisation :
clA Obj = **new** clA( );
	Déclaration :
	class clA **{**
	  private int [ ] champ = **new int** [10];
	
	  public int this [**int** index]**{**
	  get { return champ[index] ; **}**
	  set { champ[index] = ***value*** ; **}**
	  }
	}
	Utilisation :
	clA Obj = **new** clA( );

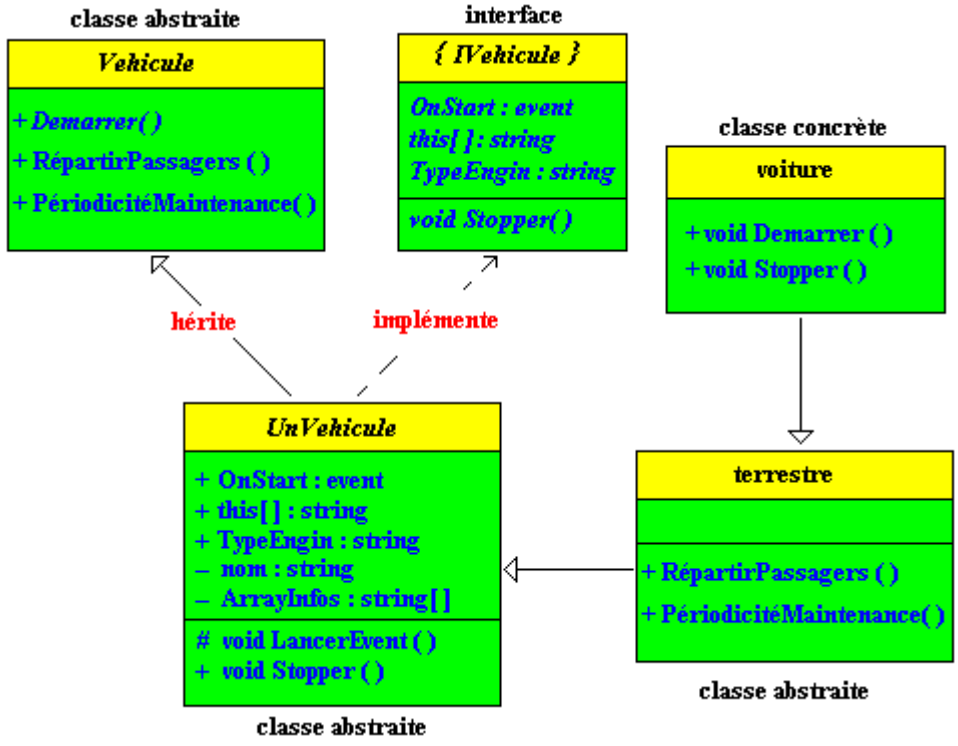
Obj.propr1 = 100 ;  **int** x = Obj.prop1 ; *// x = 100*	**for** (**int** i =0; i<5; i++) **Obj[i] = i ;**  **int** x = Obj[2] ; *// x = 2* **int** y = Obj[3] ; *// x = 3* ...

2.1.3 Paramètres

Propriété	Indexeur
Déclaration : **class** clA **{** **private int** champ; **public int** propr1**{** **get {** **return** champ*10 ; **}** **set {** champ = ***value* + *1*** ; **}** **}** **}** ***value*** est un paramètre implicite. Utilisation : clA Obj = **new** clA( ); **Obj.propr1 = 100 ;** **int** x = Obj.prop1 ; *// x = 1010*	Déclaration : **class** clA **{** **private int** [ ] champ = **new int** [10]; **public int** this [**int** k]**{** **get {** **return** champ[k]*10 ; **}** **set {** champ[k] = ***value* + *1*** ; **}** **}** **}** **k** est un paramètre formel de l'indexeur. Utilisation : clA Obj = **new** clA( ); **for** (**int** i =0; i<5; i++) **Obj[i] = i ;** **int** x = Obj[2] ; *// x = 30* **int** y = Obj[3] ; *// x = 40* ...

2.1.4 Indexeur à liaison dynamique, abstraction et interface

Reprenons l'exemple de hiérarchie de véhicules, traité avec la propriété TypeEngin de type string, en y ajoutant un indexeur de type string en proposant des définitions parallèles à la propriété et à l'indexeur :




```
abstract class  Vehicule  {  // classe abstraite mère
   ....
}
```

```
interface  IVehicule {
   ....
   string  TypeEngin    { // déclaration de propriété abstraite par défaut
     get ;
     set ;
   }
   ....
   string  this [ int k ]    { // déclaration d'indexeur abstrait par défaut
     get ;
     set ;
   }
    ....
}
```

```
  abstract class  UnVehicule : Vehicule , IVehicule  {
    private  string [ ] ArrayInfos = new  string [10] ;
    private  string  nom = "";
    ....
    public virtual  string  TypeEngin  {    // implantation virtuelle de la propriété
       get      {        return  nom ;       }
       set      {         nom = "["+value+"]" ;       }
    }
    ....
    public virtual  string  this [ int k ]   {    // implantation virtuelle de l'indexeur
       get  {  return  ArrayInfos[ k ] ;  }
       set   { ArrayInfos[ k ] = value  ;  }
```

```
    }
    ....
  }
```

```
  abstract class  Terrestre  :  UnVehicule {
   private  string  nomTerre = "";
   ....
   public override  string  TypeEngin     { // redéfinition de propriété
    get  {  return  base .TypeEngin ;  }
    set  {  string  nomTerre = value + "-Terrestre";
            base .TypeEngin = nomTerre ;  }
   }
   ....
   public override  string  this [ int k ]     { // redéfinition de  l'indexeur
    get  {  return  base[ k ] ;  }
    set  {  string  nomTerre = value + "-Terrestre" ;
            base[ k ] = nomTerre + "/set =" + k.ToString( ) + "/" ;  }
   }
  }
```

```
class Voiture  : Terrestre  {
 public override  string  TypeEngin     { // redéfinition de propriété
       get  {  return  base.TypeEngin + "-voiture";  }
      set  {   base.TypeEngin =  "(" + value + ")";  }
 }
 public override  string  this [ int n ]     { // redéfinition de  l'indexeur
       get  {  return  base[ n ] + "-voiture{get =" + n.ToString( ) + "}" ;  }
      set  {   base[ n ] =  "(" + value + ")";  }
 }
     ...
}
```

Code C# complet compilable

Code avec un événement une propriété et un indexeur

```
  public delegate void  Starting ( );    // delegate declaration de type

  abstract class  Vehicule {
    public abstract void  Demarrer ( );
    public void  RépartirPassagers ( ) {  }
   public void  PériodicitéMaintenance ( ) {  }
  }

  interface  IVehicule {
   event  Starting OnStart ;   // déclaration événement
   string  this  [ int  index]    // déclaration Indexeur
   {
    get ;
    set ;
   }
```

```
  string  TypeEngin    // déclaration propriété
  {
   get ;
   set ;
  }
 void  Stopper ( );
}

abstract class  UnVehicule : Vehicule, IVehicule {
  private  string  nom = "";
  private  string [] ArrayInfos = new  string [10] ;
  public event  Starting OnStart ; // implantation événement
  protected void  LancerEvent ( )  {
   if( OnStart  !=  null)
     OnStart ( );
  }
  public virtual  string  this  [ int  index]  { // implantation indexeur virtuel
    get  {   return  ArrayInfos[index] ;    }
   set  {    ArrayInfos[index] = value ;  }
  }
 public virtual  string  TypeEngin  { // implantation propriété virtuelle
   get  {   return  nom ;  }
   set  {    nom = "[" + value + "]";  }
 }
  public virtual void  Stopper ( ) {  }
}

abstract class  Terrestre  :  UnVehicule {
 public new void  RépartirPassagers ( ) {  }
 public new void  PériodicitéMaintenance ( ) {  }
 public override  string  this  [ int  k]    { // redéfinition indexeur
   get  {   return  base [k] ;  }
   set  {   string  nomTerre = value + "-Terrestre";
            base [k] = nomTerre + "/set = " + k.ToString () + "/";
        }
 }
 public override  string  TypeEngin   { // redéfinition propriété
   get  {   return  base .TypeEngin ;  }
   set  {   string  nomTerre = value + "-Terrestre";
             base .TypeEngin = nomTerre ;
        }
 }
}

class  Voiture  :  Terrestre {
 public override  string  this  [ int  n]    { // redéfinition indexeur
   get  {   return  base [n] + "-voiture {get = " + n.ToString ( )+ " }";  }
   set  {   string  nomTerre = value + "-Terrestre";
            base [n] = "(" + value + ")";
        }
 }
 public override  string  TypeEngin   { // redéfinition propriété
   get  {   return  base .TypeEngin + "-voiture";  }
   set  {    base .TypeEngin = "(" + value + ")";   }
 }
 public override void   Demarrer ( ) {
   LancerEvent ( );
 }
 public override void  Stopper ( ) {
   //...
  }
```

```
}


class UseVoiture
{
 static void Main ( string [] args )
 {
  // instanciation d'une voiture particulière :
  UnVehicule automobile = new Voiture ( );

  // utilisation de la propriété TypeEngin :
  automobile .TypeEngin = "Picasso";
  System .Console.WriteLine ("x est une " + automobile.TypeEngin );

  // utilisation de l'indexeur :
  automobile [0] = "Citroen";
  automobile [1] = "Renault";
  automobile [2] = "Peugeot";
  automobile [3] = "Fiat";
 for( int i = 0 ; i < 4 ; i ++ )
   System .Console.WriteLine ("Marque possible : " + automobile [i] );
  System .Console.ReadLine ( );
 }
}
```

Résultats d'exécution :

```
D:\CsharpExos\propIndex\bin\Debug\ProjPropIndex.exe
x est une [(Picasso)-Terrestre]-voiture
Marque possible : (Citroen)-Terrestre/set = 0/-voiture(get = 0)
Marque possible : (Renault)-Terrestre/set = 1/-voiture(get = 1)
Marque possible : (Peugeot)-Terrestre/set = 2/-voiture(get = 2)
Marque possible : (Fiat)-Terrestre/set = 3/-voiture(get = 3)
```

Application fenêtrées et ressources en

Plan général:

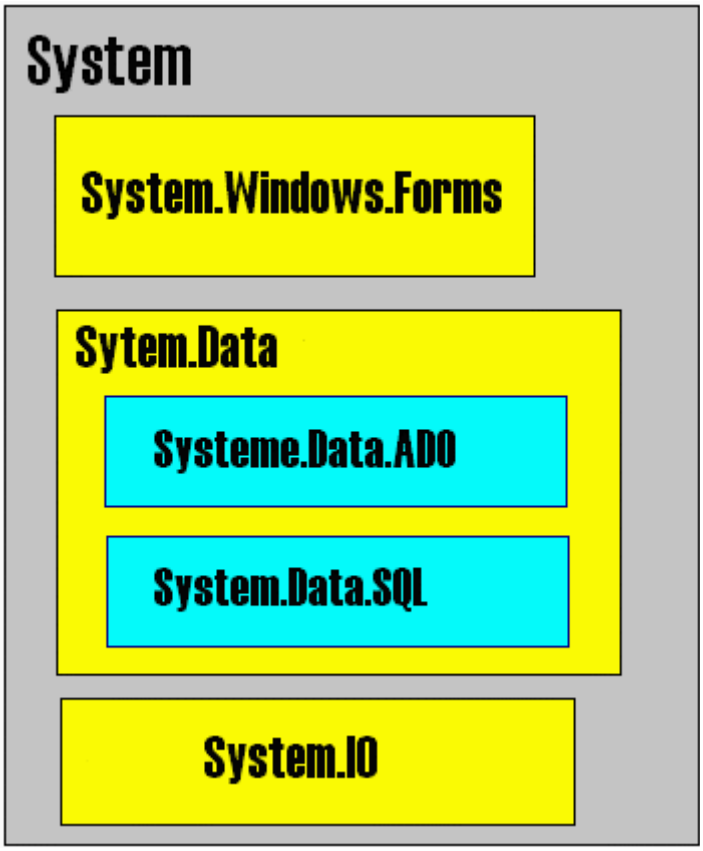
1. Les applications avec Interface Homme-Machine

1.1 Un retour sur la console
1.2 Des fenêtres à la console
1.2.1 Console et fenêtre personnalisée
1.2.2 Fenêtre personnalisée sans console
1.2.3 Que fait Application.Run ?
1.2.4 Que faire avec Application.DoEvents ?
1.3 Un formulaire en C# est une fiche
1.4 Code C# engendré par le RAD pour un formulaire
1.5 Libération de ressources non managées
1.6 Comment la libération a-t-elle lieu dans le NetFrameWork ?
1.7 Peut-on influer sur cette la libération dans le NetFrameWork ?
1.8 Design Pattern de libération des ressources non managées
1.9 Un exemple utilisant la méthode Dispose d'un formulaire
1.10 L'instruction USING appelle Dispose( )
1.11 L'attribut [STAThread]

1. Les applications avec Interface Homme-Machine

Les exemples et les traitement qui suivent sont effectués sous l'OS windows depuis la version NetFrameWork 1.1, les paragraphes 1.3, 1.4, ... , 1.10 expliquent le contenu du code généré automatiquement par Visual Studio ou C# Builder de Borland Studio pour développer une application fenêtrée.

Le NetFrameWork est subdivisé en plusieurs espaces de nom, l'espace de noms System contient plusieurs classes, il est subdivisé lui-même en plusieurs sous-espaces de noms :



etc ...

L'espace des noms **System.Windows.Forms** est le domaine privilégié du NetFrameWork dans lequel l'on trouve des classes permettant de travailler sur des applications fenêtrées.

La classe **Form** de l'espace des noms **System.Windows.Forms** permet de créer une fenêtre classique avec barre de titre, zone client, boutons de fermeture, de zoom...

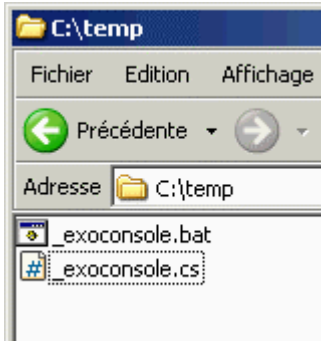
En C#, Il suffit d'instancier un objet de cette classe pour obtenir une fenêtre classique qui s'affiche sur l'écran.

1.1 un retour sur la console

Les exemples sont écrits pour la version 1.1 mais restent valides pour les versions ultérieures.

Le code C# peut tout comme le code Java être écrit avec un éditeur de texte rudimentaire du style bloc-note, puis être compilé directement à la **console** par appel au compilateur **csc.exe**.

Soient par exemple dans un dossier temp du disque C: , deux fichiers :



Le fichier "**_exoconsole.bat**" contient la commande système permettant d'appeller le compilateur C#.
Le fichier "**_exoconsole.cs**" le programme source en C#.

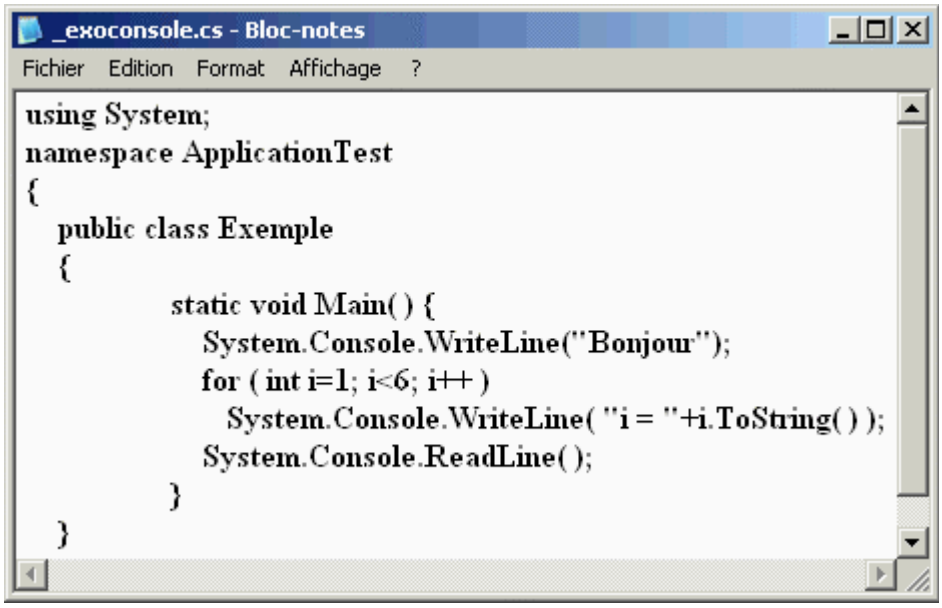
Construction de la commande de compilation "**_exoconsole.bat**" :

On indique d'abord le chemin (variable path) du répertoire où se trouve le compilateur **csc.exe**, puis on lance l'appel au compilateur avec ici , un nombre minimal de paramètres :

Attributs et paramètres de la commande	fonction associée
set path = C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322	Le chemin absolu permettant d'accéser au dossier contenant le compilateur C# (**csc.exe**)
/t:exe	Indique que nous voulons engendrer une exécutable console (du code MSIL)
/out: _exo.exe	Indique le nom que doit porter le fichier exécutable MSIL après compilation
_exoconsole.cs	Le chemin complet du fichier source C# à compiler (ici il est dans le même répertoire que la commande, seul le nom du fichier suffit)

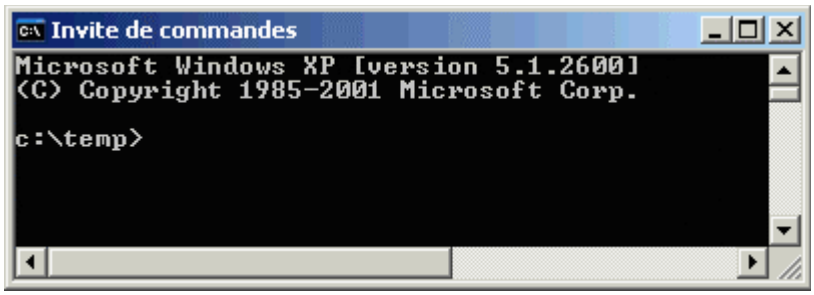
Texte de la commande dans le Bloc-note :

Nous donnons le contenu du fichier source **_exoconsole.cs** à compiler :

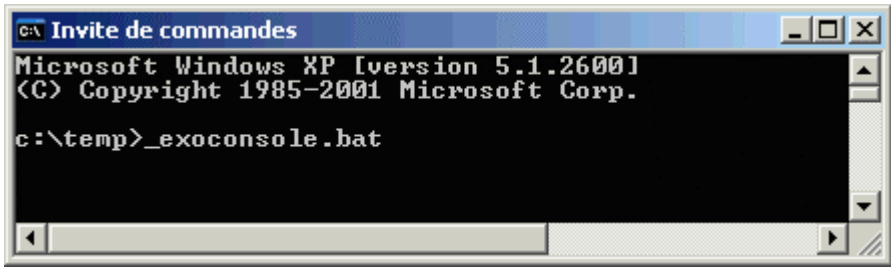


Le programme précédent affiche le mot Bonjour suivit de l'exécution de la boucle sur 5 itérations.

On lance l'invite de commande de Windows ici dans le répertoire c:\temp :



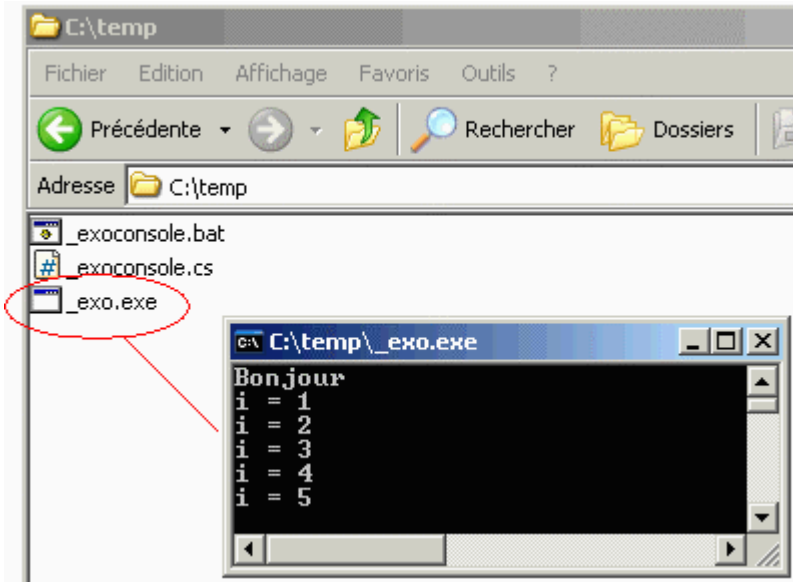
On tape au clavier et l'on exécute la commande "**_exoconsole.bat**" :



Elle appelle le compilateur **csc.exe** qui effectue la compilation du programme **_exoconsole.cs** sans signaler d'erreur particulière et qui a donc engendré une fichier MSIL nommé **_exo.exe** :

```
Invite de commandes
Copyright (C) Microsoft Corporation 2001-2002. Tous droits réservés.

c:\temp>_exoconsole.bat

c:\temp>set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322

c:\temp>csc /t:exe /out:_exo.exe   _exoconsole.cs
Compilateur Microsoft (R) Visual C# .NET version 7.10.3052.4
pour Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. Tous droits réservés.

c:\temp>_
```

Lançons par un double click l'exécution du programme **_exo.exe** qui vient d'être engendré :

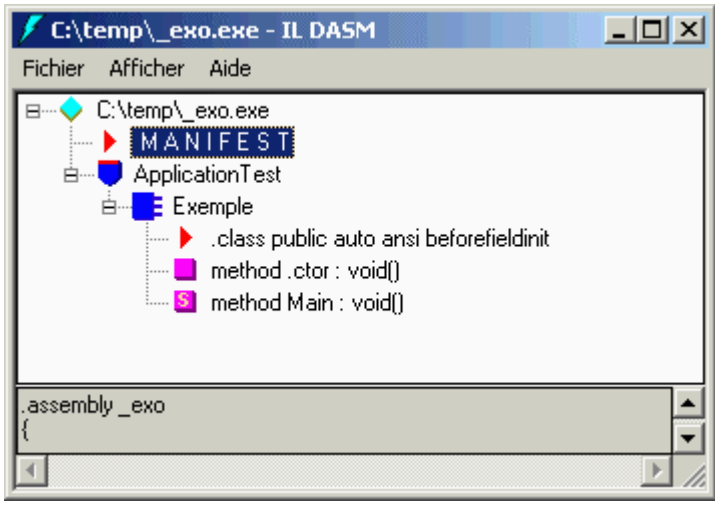


Nous constatons que l'exécution par le CLR du fichier **_exo.exe** a produit le résultat escompté c'est à dire l'affichage du mot Bonjour suivit de l'exécution de la boucle sur 5 itérations.

Afin que le lecteur soit bien convaincu que nous sommes sous NetFramework et que les fichiers exécutables ne sont pas du binaire exécutable comme jusqu'à présent sous Windows, mais des fichiers de code MSIL exécutable par le CLR, nous passons le fichier **_exo.exe** au désassembleur **ildasm** par la commande "ildasm.bat".

Le désassembleur MSIL Disassembler (Ildasm.exe) est un utilitaire inclus dans le kit de développement .NET Framework SDK, il est de ce fait utilisable avec tout langage de .Net dont C#. ILDasm.exe analyse toutes sortes d'assemblys .NET Framework **.exe** ou **.dll** et présente les informations dans un format explicite. Cet outil affiche bien plus que du code MSIL (Microsoft Intermediate Language) ; il présente également les espaces de noms et les types, interfaces comprises.

Voici l'inspection du fichier **_exo.exe** par **ildasm** :



Demandons à **ildasm** l'inspection du code MSIL engendré pour la méthode Main( ) :

*Nous avons mis en **gras et en italique** les commentaires d'instructions sources*

Exemple::methodeMain void( )

```
.method private hidebysig static void  Main( ) cil managed
{
  .entrypoint
  // Code size       51 (0x33)
  .maxstack  2
  .locals init ([0] int32 i)
  .language '{3F5162F8-07C6-11D3-9053-00C04FA302A1}', '{994B45C4-E6E9-11D2-903F-
00C04FA302A1}', '{5A869D0B-6611-11D3-BD2A-0000F80849BD}'
// Source File 'c:\temp\_exoconsole.cs'
//000007:        System.Console.WriteLine("Bonjour");
  IL_0000:  ldstr      "Bonjour"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
//000008:        for ( int i=1; i<6; i++ )
  IL_000a:  ldc.i4.1
  IL_000b:  stloc.0
  IL_000c:  br.s       IL_0028
//000009:          System.Console.WriteLine( "i = "+i.ToString( ) );
  IL_000e:  ldstr      "i = "
  IL_0013:  ldloca.s   i
  IL_0015:  call       instance string [mscorlib]System.Int32::ToString()
  IL_001a:  call       string [mscorlib]System.String::Concat(string,string)
  IL_001f:  call       void [mscorlib]System.Console::WriteLine(string)
//000008:        for ( int i=1; i<6; i++ )
  IL_0024:  ldloc.0
  IL_0025:  ldc.i4.1
  IL_0026:  add
  IL_0027:  stloc.0
  IL_0028:  ldloc.0
  IL_0029:  ldc.i4.6
  IL_002a:  blt.s      IL_000e
//000009:          System.Console.WriteLine( "i = "+i.ToString( ) );
//000010:        System.Console.ReadLine( );
  IL_002c:  call       string [mscorlib]System.Console::ReadLine()
  IL_0031:  pop
```

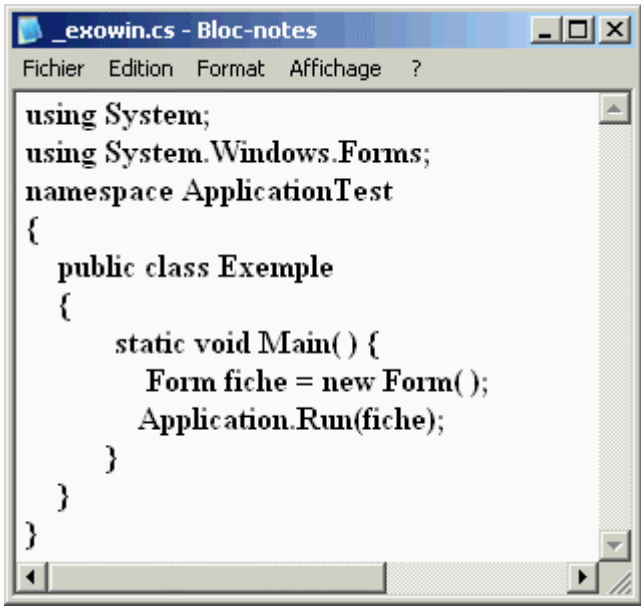
```
//000011:    }
  IL_0032:  ret
} // end of method Exemple::Main
```

1.2 Des fenêtres à la console

On peut donc de la même façon compiler et exécuter à partir de la console, des programmes C# contenant des fenêtres, comme en java il est possible d'exécuter à partir de la console des applications contenant des Awt ou des Swing, idem en Delphi. Nous proposons au lecteur de savoir utiliser des programmes qui allient la console à une fenêtre classique, ou des programmes qui ne sont formés que d'une fenêtre classique (à minima). Ici aussi, les exemples sont écrits en version 1.1 mais sont valides pour les versions ultérieures.

1.2.1 fenêtre console et fenêtre personnalisée ensembles

Ecrivons le programme C# suivant dans un fichier que nous nommons "**_exowin.cs**" :



Ce programme "**_exowin.cs**" utilise la classe Form et affiche une fenêtre de type Form :

La première instruction instancie un objet nommé **fiche** de la classe **Form**
Form fiche = new Form( ) ;

La fenêtre **fiche** est ensuite "initialisée" et "lancée" par l'instruction
Application.Run(fiche) ;

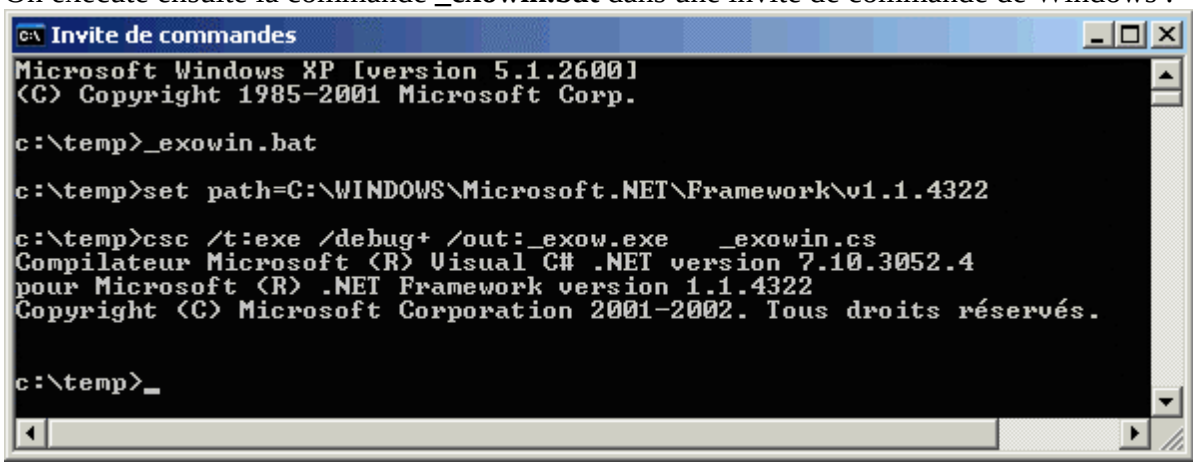
On construit une commande nommée **_exowin.bat**, identique à celle du paragraphe précédent, afin de lancer la compilation du programme **_exowin.cs**, nous décidons de nommer **_exow.exe**

l'exécutable MSIL obtenu après compilation.

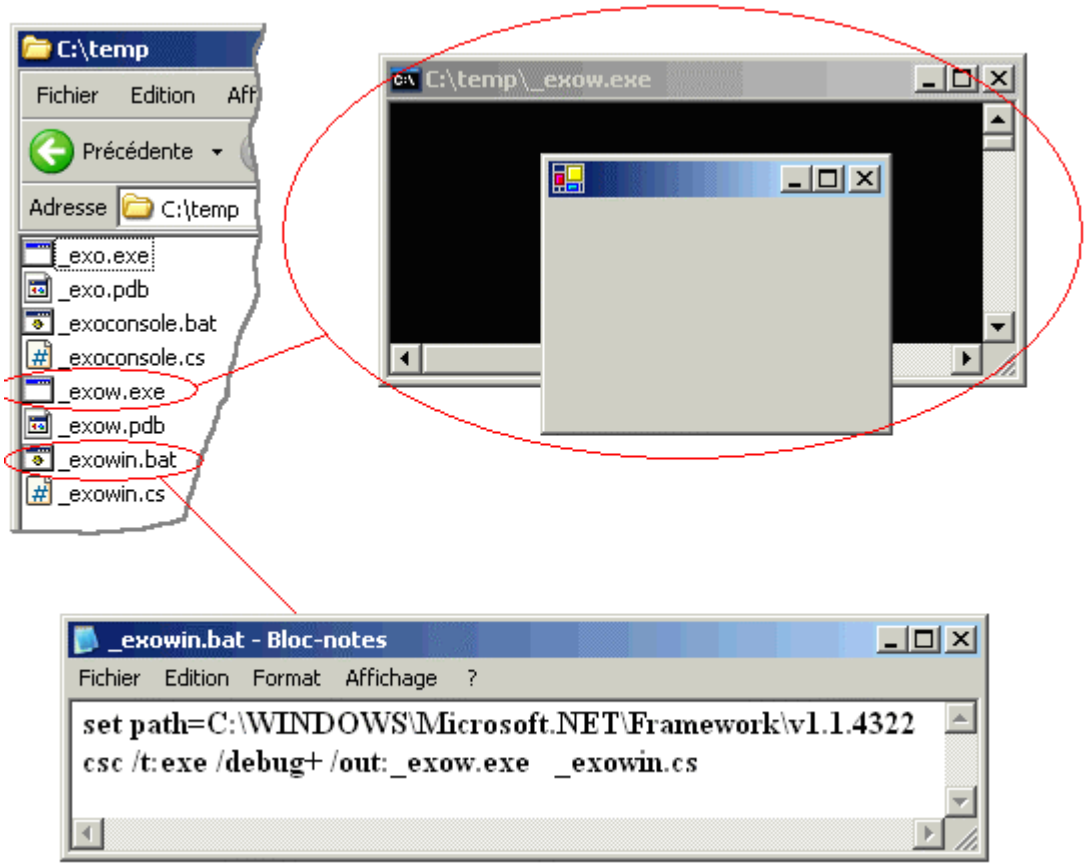
contenu fichier de commande **_exowin.bat** :

set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
csc /t:exe /out:_exow.exe _exowin.cs

On exécute ensuite la commande **_exowin.bat** dans une invite de commande de Windows :



Cette commande a généré comme précédemment l'exécutable MSIL nommé ici **_exow.exe** dans le dossier c:\temp, nous exécutons le programme **_exow.exe** et nous obtenons l'affichage d'une fenêtre de console et de la fenêtre fiche :



Remarque:

> L'attribut **/debug+** rajouté ici, permet d'engendrer un fichier **_exo.pdb** qui contient des informations de déboguage utiles à ildasm.

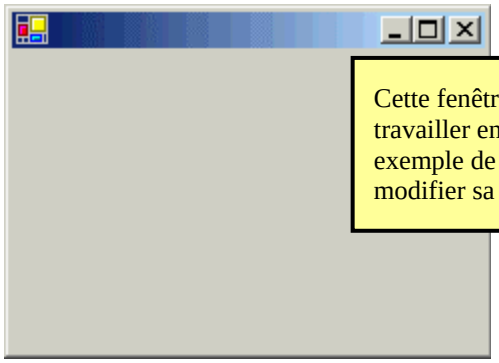
1.2.2 fenêtre personnalisée sans fenêtre console

Si nous ne voulons pas voir apparaître de fenêtre de console mais seulement la fenêtre fiche, il faut alors changer dans le paramétrage du compilateur l'attribut **target.** De la valeur **csc /t:exe** il faut passer à la valeur **csc /t:winexe** :

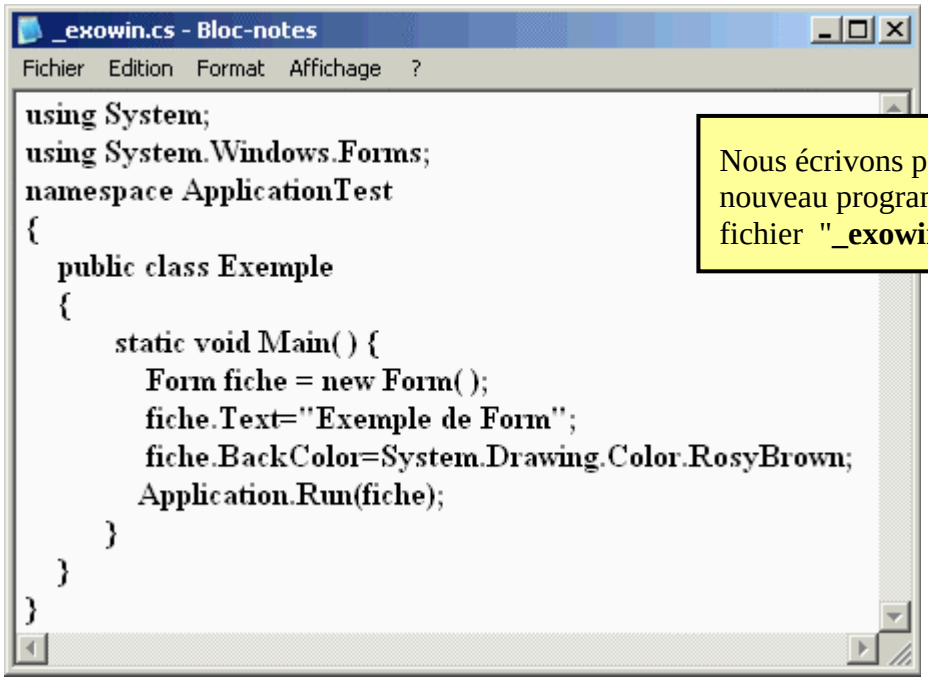
Nouveau fichier de commande **_exowin.bat** :

```
set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
csc /t:winexe  /out:_exow.exe   _exowin.cs
```

Nous compilons en lançant la nouvelle commande **_exowin.bat** puis nous exécutons le nouveau programme **_exow.exe**. Nous obtenons cette fois-ci l'affichage d'une seule fenêtre (la fenêtre de console a disparu) :

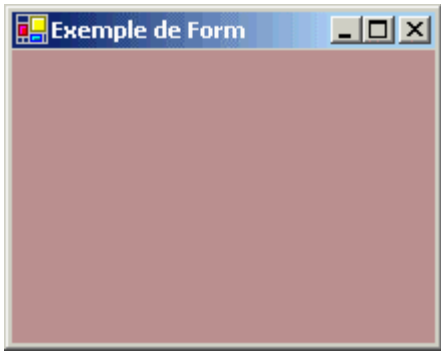


Cette fenêtre est un peu trop terne, nous pouvons travailler en mode console sur la fiche Form, afin par exemple de mettre un texte dans la barre de titre et de modifier sa couleur de fond.



Nous écrivons pour cela le nouveau programme C# dans le fichier "**_exowin.cs**"

Nous compilons en lançant la commande **_exowin.bat** puis nous exécutons le nouveau programme **_exow.exe** et nous obtenons l'affichage de la fenêtre fiche avec le texte "Exemple de Form" dans la barre de titre et sa couleur de fond marron-rosé (Color.RosyBrown) :



Consultons à titre informatif avec **ildasm** le code MSIL engendré pour la méthode Main( ) :

*Nous avons mis en **gras et en italique** les commentaires d'instructions sources*

Exemple::methodeMain void( )

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       35 (0x23)
  .maxstack  2
  .locals init ([0] class [System.Windows.Forms]System.Windows.Forms.Form fiche)
  .language '{3F5162F8-07C6-11D3-9053-00C04FA302A1}', '{994B45C4-E6E9-11D2-903F-00C04FA302A1}',
'{5A869D0B-6611-11D3-BD2A-0000F80849BD}'
// Source File 'c:\temp\_exowin.cs'
//000008:              Form fiche = new Form( );
  IL_0000:  newobj     instance void [System.Windows.Forms]System.Windows.Forms.Form::.ctor()
  IL_0005:  stloc.0
//000009:              fiche.Text="Exemple de Form";
  IL_0006:  ldloc.0
  IL_0007:  ldstr      "Exemple de Form"
  IL_000c:  callvirt   instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
//000010:              fiche.BackColor=System.Drawing.Color.RosyBrown;
  IL_0011:  ldloc.0
  IL_0012:  call       valuetype [System.Drawing]System.Drawing.Color
[System.Drawing]System.Drawing.Color::get_RosyBrown()
  IL_0017:  callvirt   instance void
[System.Windows.Forms]System.Windows.Forms.Control::set_BackColor(valuetype
[System.Drawing]System.Drawing.Color)
//000011:              Application.Run(fiche);
  IL_001c:  ldloc.0
  IL_001d:  call       void [System.Windows.Forms]System.Windows.Forms.Application::Run(class
[System.Windows.Forms]System.Windows.Forms.Form)
//000012:          }
  IL_0022:  ret
} // end of method Exemple::Main
```

1.2.3 Que fait Application.Run(fiche)

Comme les fenêtres dans Windows ne sont pas des objets ordinaires, pour qu'elles fonctionnent correctement vis à vis des messages échangés entre la fenêtre et le système, il est nécessaire de lancer une boucle d'attente de messages du genre :

```
tantque non ArrêtSysteme faire
  si événement alors
     construire Message ;
     si Message ≠ ArrêtSysteme alors
        reconnaître la fenêtre à qui est destinée ce Message;
       distribuer ce Message
     fsi
  fsi
ftant
```

La documentation technique indique que l'une des surcharges de la méthode de classe **Run** de la classe Application "**public static void** Run(**Form** mainForm);" *exécute une boucle de messages d'application standard sur le thread en cours et affiche la Form spécifiée*. Nous en déduisons que notre fenêtre fiche est bien initialisée et affichée par cette méthode Run.

Observons à contrario ce qui se passe si nous n'invoquons pas la méthode **Run** et programmons l'affichage de la fiche avec sa méthode **Show** :

```
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( );
        }
   }
}
```

Lorsque nous compilons puis exécutons ce programme la fiche apparaît correctement (titre et couleur) d'une manière fugace car elle disparaît aussi vite qu'elle est apparue. En effet le programme que nous avons écrit est correct :

Ligne d'instruction du programme	Que fait le CLR
{	il initialise l'exécution de la méthode Main
Form fiche = **new** Form( );	il instancie une Form nommée fiche
fiche.Text="Exemple de Form";	il met un texte dans le titre de fiche
fiche.BackColor=System.Drawing.Color.RosyBrown;	il modifie la couleur du fond de fiche

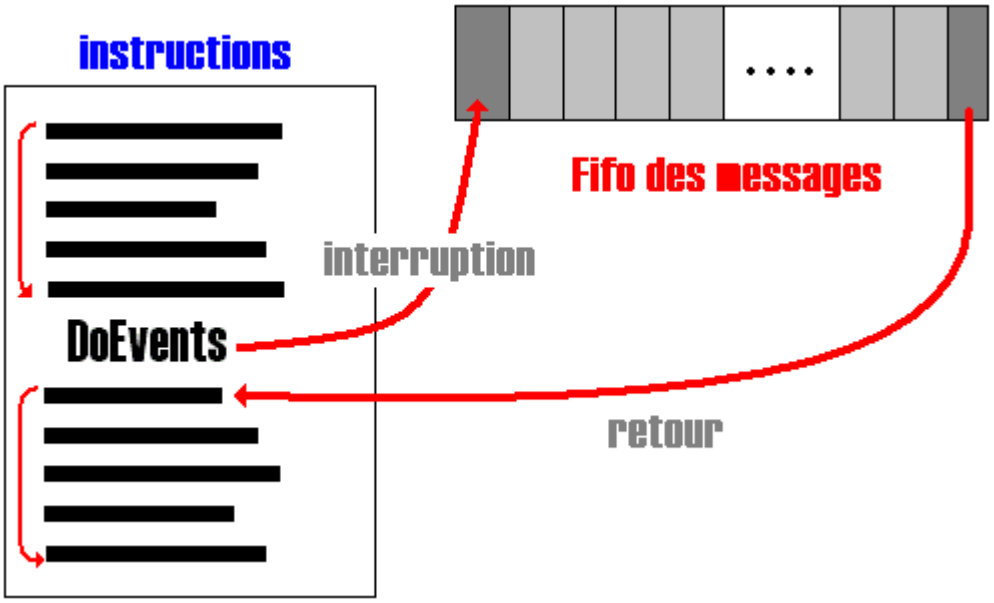
fiche.Show( );	il rend la fiche visible
}	fin de la méthode Main et donc tout est détruit et libéré et le processus est terminé.

La fugacité de l'affichage de notre fenêtre fiche est donc normale, puisqu'à peine créée la fiche a été détruite.

Si nous voulons que notre objet de fiche persiste sur l'écran, il faut simuler le comportement de la méthode classe **Run**, c'est à dire qu'il nous faut écrire une boucle de messages.

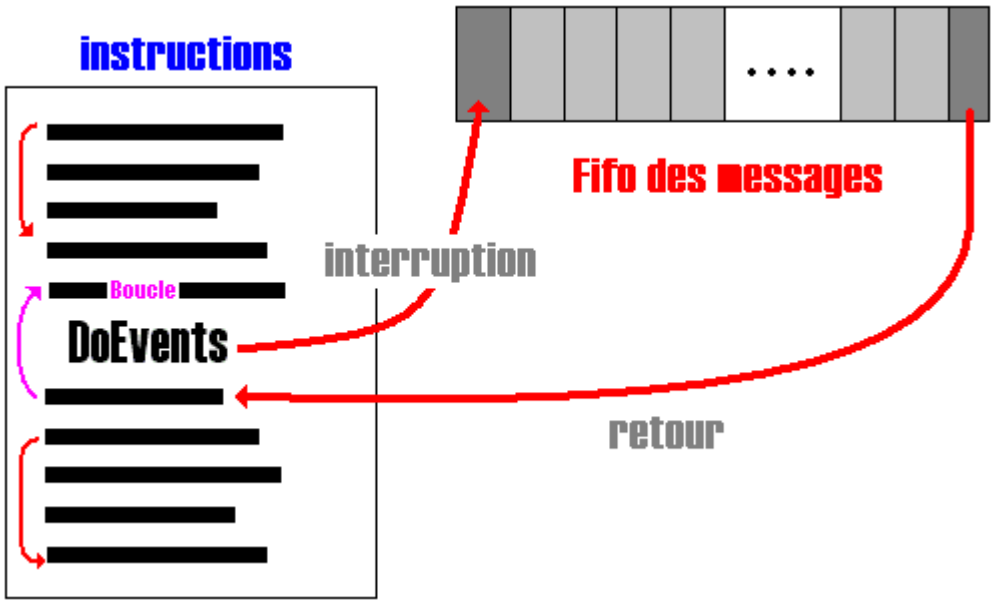
1.2.4 Que faire avec Application.DoEvents( )

Nous allons utiliser la méthode de classe **DoEvents( )** de la classe **Application** qui existe depuis Visual Basic 2, et qui permet de traiter tous les messages Windows présents dans la file d'attente des messages de la fenêtre (*elle passe la main au système d'une façon synchrone*) puis revient dans le programme qui l'a invoquée (identique à processMessages de Delphi).



Nous créeons artificiellement une boucle en apparence infinie qui laisse le traitement des messages s'effectuer et qui attend qu'on lui demande de s'arrêter par l'intermédiaire d'un booléen stop dont la valeur change par effet de bord grâce à **DoEvents** :

```
static bool stop = false;
while (!stop) Application.DoEvents( );
```



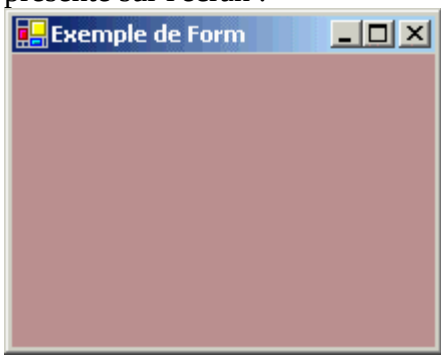
Il suffit que lorsque **DoEvents( )** s'exécute l'une des actions de traitement de messages provoque la mise du booléen stop à true pour que la boucle s'arrête et que le processus se termine.

Choisissons une telle action par exemple lorsque l'utilisateur clique sur le bouton de fermeture de la fiche, la fiche se ferme et l'événement closed est déclenché, **DoEvents( )** revient dans la boucle d'attente **while** (!stop) Application.DoEvents( ); au tour de boucle suivant. Si lorsque l'événement close de la fiche a lieu nous en profitons pour mettre le booléen stop à true, dès le retour de **DoEvents( )** le prochain tour de boucle arrêtera l'exécution de la boucle et le corps d'instruction de **Main** continuera à s'exécuter séquentiellement jusqu'à la fin (ici on arrêtera le processus).

Ci-dessous le programme C# à mettre dans le fichier "**_exowin.cs**" :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
   public class Exemple   {
        static bool stop = false; // le drapeau d'arrêt de la boucle d'attente

         static void  fiche_Closed (object sender, System.EventArgs e)  {
        // le gestionnaire de l'événement closed de la fiche
            stop = true;
         }

        static void Main( ) {
            System.Windows.Forms.Button button1 = new System.Windows.Forms.Button( );
            Form fiche = new Form( );
            fiche.Text="Exemple de Form";
            fiche.BackColor=System.Drawing.Color.RosyBrown;
            fiche.Closed += new System.EventHandler(fiche_Closed); // abonnement du gestionnaire
            fiche.Show( );
            while (!stop) Application.DoEvents( ); // boucle d'attente
           //... suite éventuelle du code avant arrêt
        }
   }
}
```

Lorsque nous compilons puis exécutons ce programme la fiche apparaît correctement et reste présente sur l'écran :



Elle se ferme et disparaît lorsque nous cliquons sur le bouton de fermeture.

On peut aussi vouloir toujours en utilisant la boucle infinie qui laisse le traitement des messages s'effectuer ne pas se servir d'un booléen et continuer après la boucle, mais plutôt essayer d'interrompre et de terminer l'application directement dans la boucle infinie sans exécuter la suite du code. La classe Application ne permet pas de terminer le processus.

Attention à l'utilisation de la méthode Exit de la classe Application qui semblerait être utilisable dans ce cas, en effet cette méthode arrête toutes les boucles de messages en cours sur tous les threads et ferme toutes les fenêtres de l'application; mais cette méthode ne force pas la fermeture de l'application.

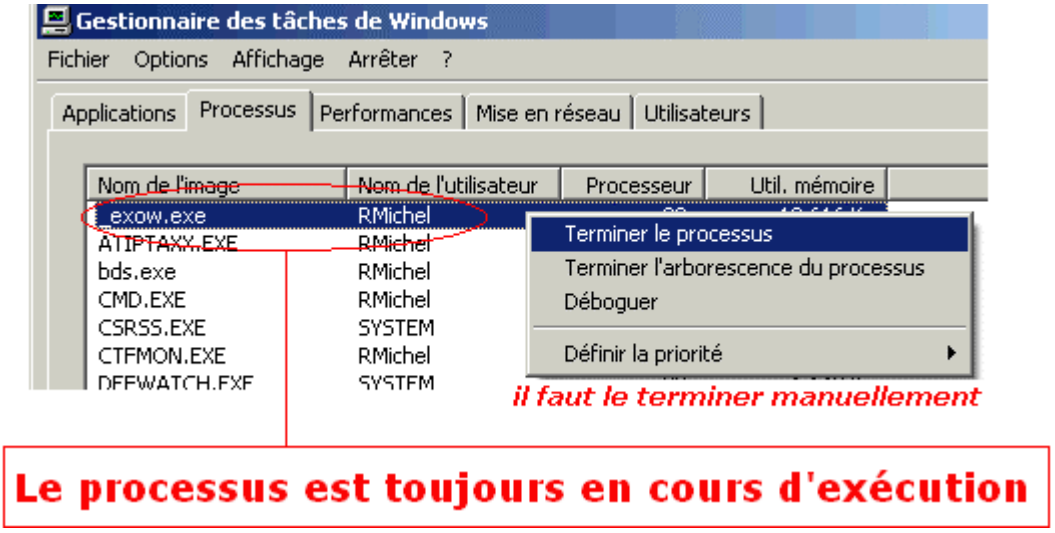
Pour nous en convaincre compilons et exécutons le programme ci-après dans lequel l'événement fiche_Closed appelle Application.Exit( )

Ci-dessous le programme C# à mettre dans le fichier "**_exowin.cs**" :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
   public class Exemple
   {
        static void  fiche_Closed (object sender, System.EventArgs e)
        { // le gestionnaire de l'événement closed de la fiche
           Application.Exit( ); // on ferme la fenêtre, mais on ne termine pas le processus
         }

        static void Main( ) {
           System.Windows.Forms.Button button1 = new System.Windows.Forms.Button( );
           Form fiche = new Form( );
           fiche.Text="Exemple de Form";
           fiche.BackColor=System.Drawing.Color.RosyBrown;
           fiche.Closed += new System.EventHandler(fiche_Closed); // abonnement du gestionnaire
           fiche.Show( );
           while (true) Application.DoEvents( ); // boucle infinie
        }
   }
}
```

Lorsque nous compilons puis exécutons ce programme la fiche apparaît correctement et reste

présente sur l'écran, puis lorsque nous fermons la fenêtre comme précédemment, elle disparaît, toutefois le processus _exow.exe est toujours actif (la boucle tourne toujours, mais la fenêtre a été fermée) en faisant apparaître le gestionnaire des tâches de Windows à l'onglet processus nous voyons qu'il est toujours présent dans la liste des processus actifs. Si nous voulons l'arrêter il faut le faire manuellement comme indiqué ci-dessous :



Comment faire pour réellement tout détruire ?

Il faut pouvoir détruire le processus en cours (en prenant soin d'avoir tout libéré avant si nécessaire), pour cela le NetFrameWork dispose d'une classe Process qui permet *l'accès à des processus locaux ainsi que distants, et vous permet de démarrer et d'arrêter des processus systèmes locaux*.

Nous pouvons connaître le processus en cours d'activation (ici, c'est notre application_exow.exe) grâce à la méthode de classe GetCurrentProcess et nous pouvons "tuer" un processus grâce à la méthode d'instance Kill :

```
static void  fiche_Closed (object sender, System.EventArgs e)
{ // le gestionnaire de l'événement closed de la fiche
        System.Diagnostics.Process ProcCurr = System.Diagnostics.Process.GetCurrentProcess( );
        ProcCurr.Kill( );
}
```

Ci-dessous le programme C# à mettre dans le fichier "**_exowin.cs**" :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
   public class Exemple
   {
        static void  fiche_Closed (object sender, System.EventArgs e)
        { // le gestionnaire de l'événement closed de la fiche
           System.Diagnostics.Process ProcCurr = System.Diagnostics.Process.GetCurrentProcess( );
           ProcCurr.Kill( );
         }

        static void Main( ) {
```

```
            System.Windows.Forms.Button button1 = new System.Windows.Forms.Button( );
            Form fiche = new Form( );
            fiche.Text="Exemple de Form";
            fiche.BackColor=System.Drawing.Color.RosyBrown;
            fiche.Closed += new System.EventHandler(fiche_Closed); // abonnement du gestionnaire
            fiche.Show( );
            while (true) Application.DoEvents( ); // boucle infinie
        }
    }
}
```

Après compilation, exécution et fermeture en faisant apparaître le gestionnaire des tâches de Windows à l'onglet processus nous voyons que le processus a disparu de la liste des processus actifs du système. Nous avons donc bien interrompu la boucle infinie.

Toutefois la console n'est pas l'outil préférentiel de C# dans le sens où C# est l'outil de développement de base de .Net et que cette architecture a vocation à travailler essentiellement avec des fenêtres.

Dans ce cas nous avons tout intérêt à utiliser un RAD visuel C# pour développer ce genre d'applications (comme l'on utilise Delphi pour le développement d'IHM en pascal objet). Une telle utilisation nous procure le confort du développement visuel, la génération automatique d'une bonne partie du code répétitif sur une IHM, l'utilisation et la réutilisation de composants logiciels distribués sur le net.

RAD utilisables

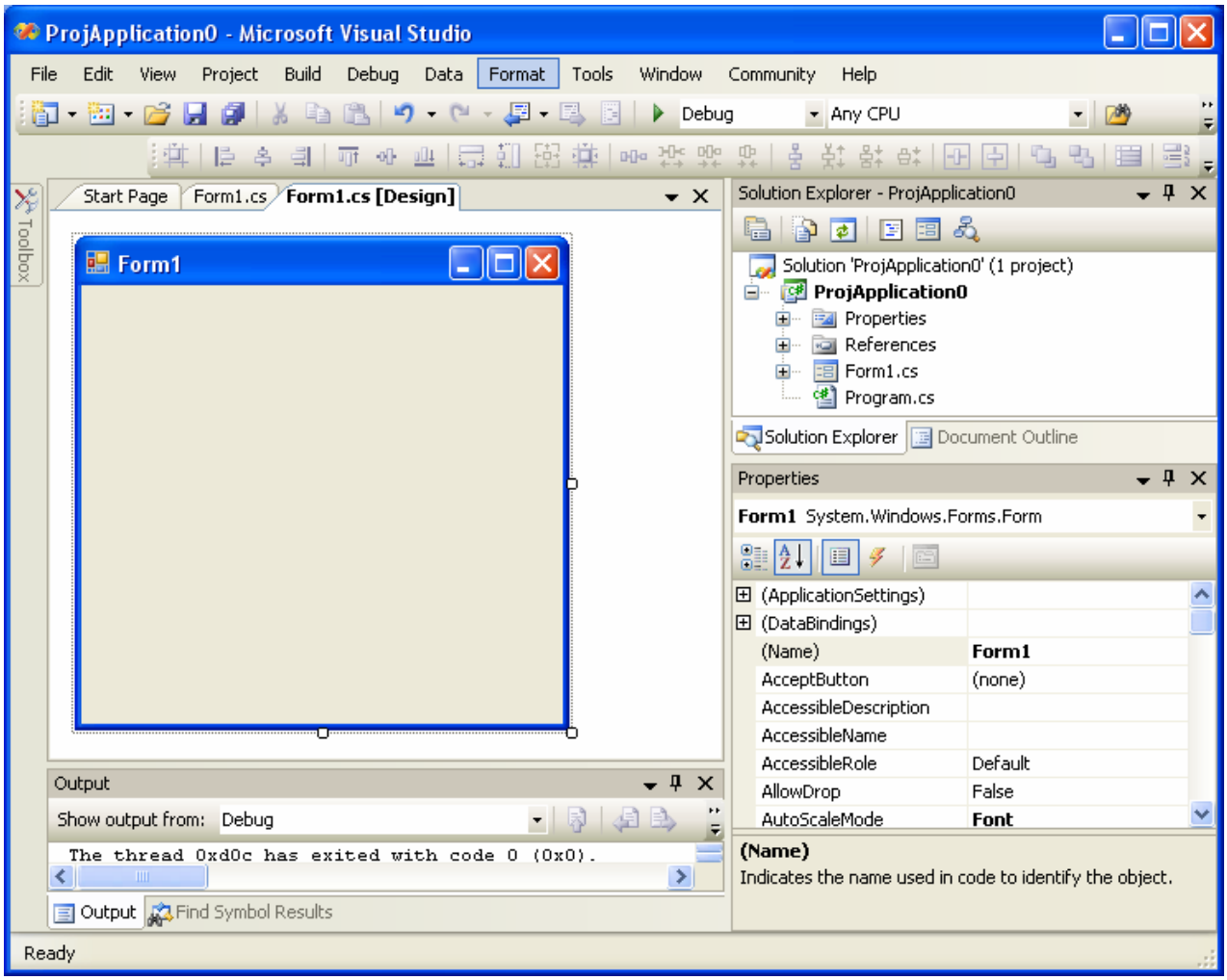
- **Visual Studio de microsoft** contient deux RAD de développement pour .Net, VBNet (fondé sur Visual Basic réellement objet et entièrement rénové) et Visual C# (fondé sur le langage C#), parfaitement adapté à .Net. *(versions express gratuites)*
- **C# Builder de Borland** reprenant les fonctionnalités de Visual C# inclus dans Borland Studio 2006 *(versions personnelle gratuite)*
- **Le monde de l'open source** construit un produit nommé **sharpDevelop** fournit à tous gratuitement aussi les mêmes fonctions que Visuel C# express de Microsoft, mais il est généralement en retard sur les versions de Microsoft.

1.3 un formulaire en C# est une fiche

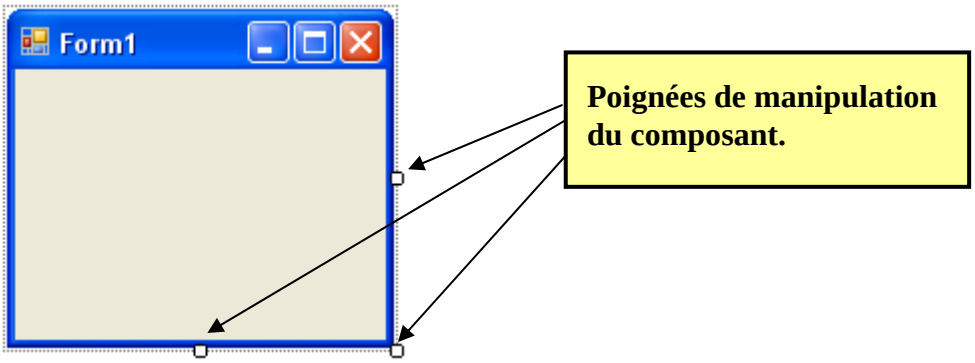
Les fiches ou formulaires C# représentent l'interface utilisateur (IHM) d'une application sous l'apparence d'une fenêtre visuelle. Comme les deux environnements RAD, Visual studio C# de Microsoft et C# Builder de Borland permettent de concevoir visuellement des applications avec IHM, nous dénommerons l'un ou l'autre par le terme général RAD C#.

Etant donné la disponibilité gratuite des trois RAD cités plus haut, dans cette partie de l'ouvrage nous avons utilisé C# Builder en version personnelle (très suffisante pour déjà écrire de bonnes applications) nous illustrerons donc tous nos exemples avec ce RAD. Il est évident que si le lecteur utilise un autre RAD, il ne sera pas dépaysé car les ergonomies de ces 3 RAD sont très proches.

Voici l'apparence d'un formulaire (ou fiche) dans le RAD Visuel C# en mode conception :



La fiche elle-même est figurée par l'image ci-dessous retaillable à volonté à partir de cliqué glissé sur l'une des huits petites "poignées carrées" situées aux points cardinaux de la fiche :



Ces formulaires sont en faits des objets d'une classe nommée **Form** de l'espace des noms **System.Windows.Forms**. Ci-dessous la hiérarchie d'héritage de Object à Form :

System.Object

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

La classe Form est la classe de base de tout style de fiche (ou formulaire) à utiliser dans votre application (statut identique à TForm dans Delphi) : fiche de dialogue, sans bordure etc..

Les différents styles de fiches sont spécifiés par l'énumération FormBorderStyle :

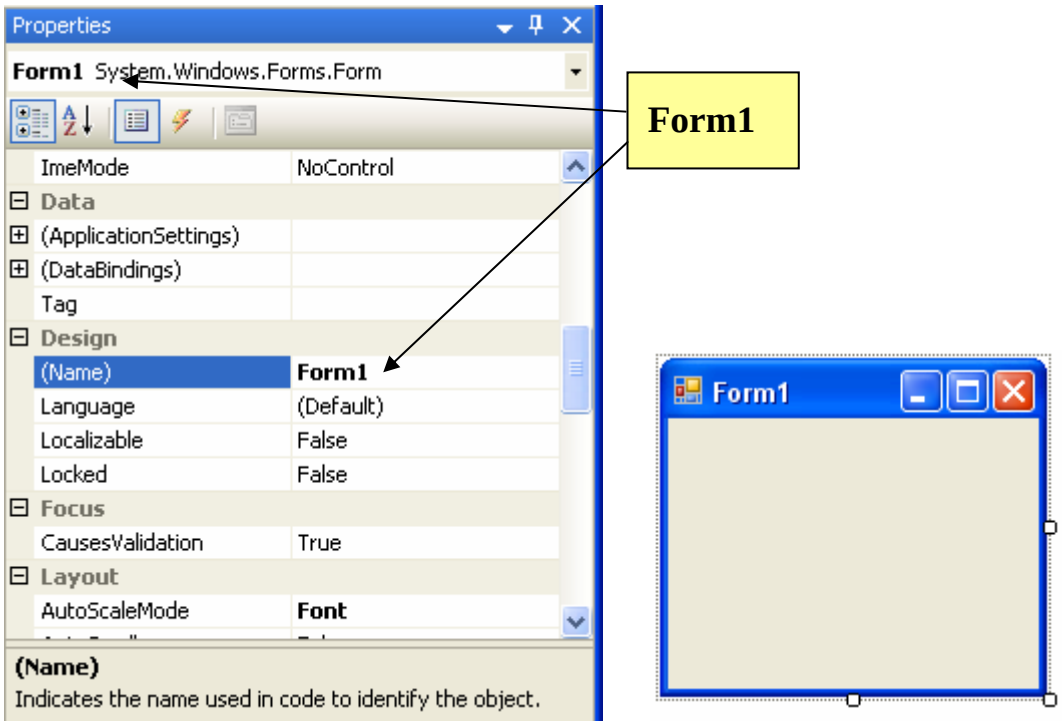
```
public Enum FormBorderStyle {Fixed3D, FixedDialog, FixedSingle, FixedToolWindow,
None, Sizable, SizableToolWindow  }
```

Dans un formulaire, le style est spécifié par la **propriété** FormBorderStyle de la classe Form :

```
public FormBorderStyle FormBorderStyle {get; set;}
```

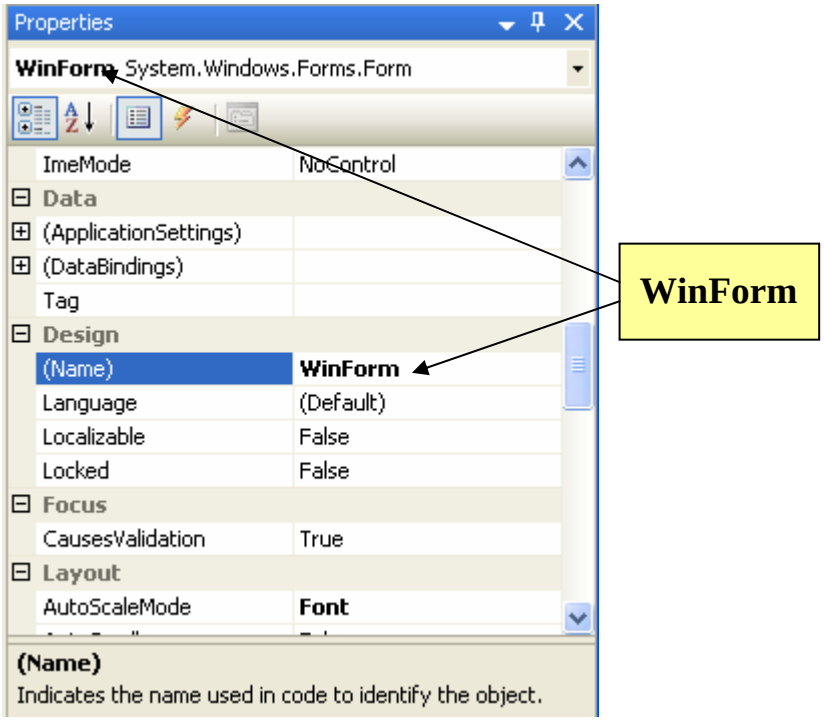
Toutes les propriétés en lecture et écriture d'une fiche sont accessibles à travers l'inspecteur d'objet qui répercute immédiatement en mode conception toute modification. Certaines provoquent des changements visuels d'autres non :

1°)Nous changeons le nom d'identificateur de notre fiche nommée **Form1** dans le programme en modifiant sa propriété Name dans l'inspecteur d'objet :

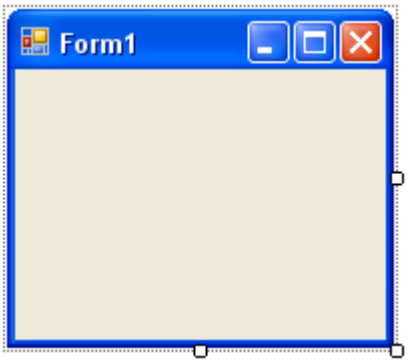


Le RAD construit automatiquement notre fiche principale comme une classe héritée de la classe Form et l'appelle Form1 : **public class** WinForm : System.Windows.Forms.Form{ ... }

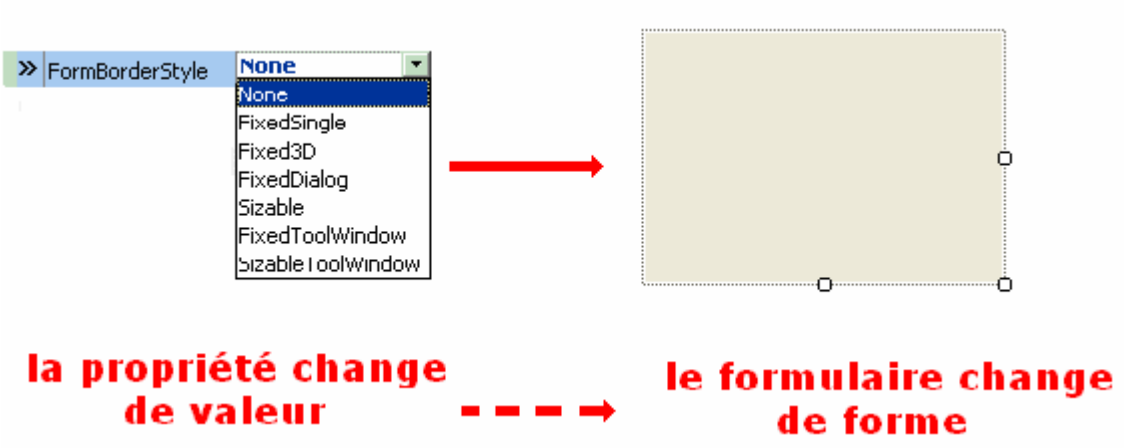
Après modification de la propriété Name par exemple par le texte **WinForm**, nous obtenons :



La classe de notre formulaire s'appelle désormais **WinForm**, mais son aspect visuel est resté le même :



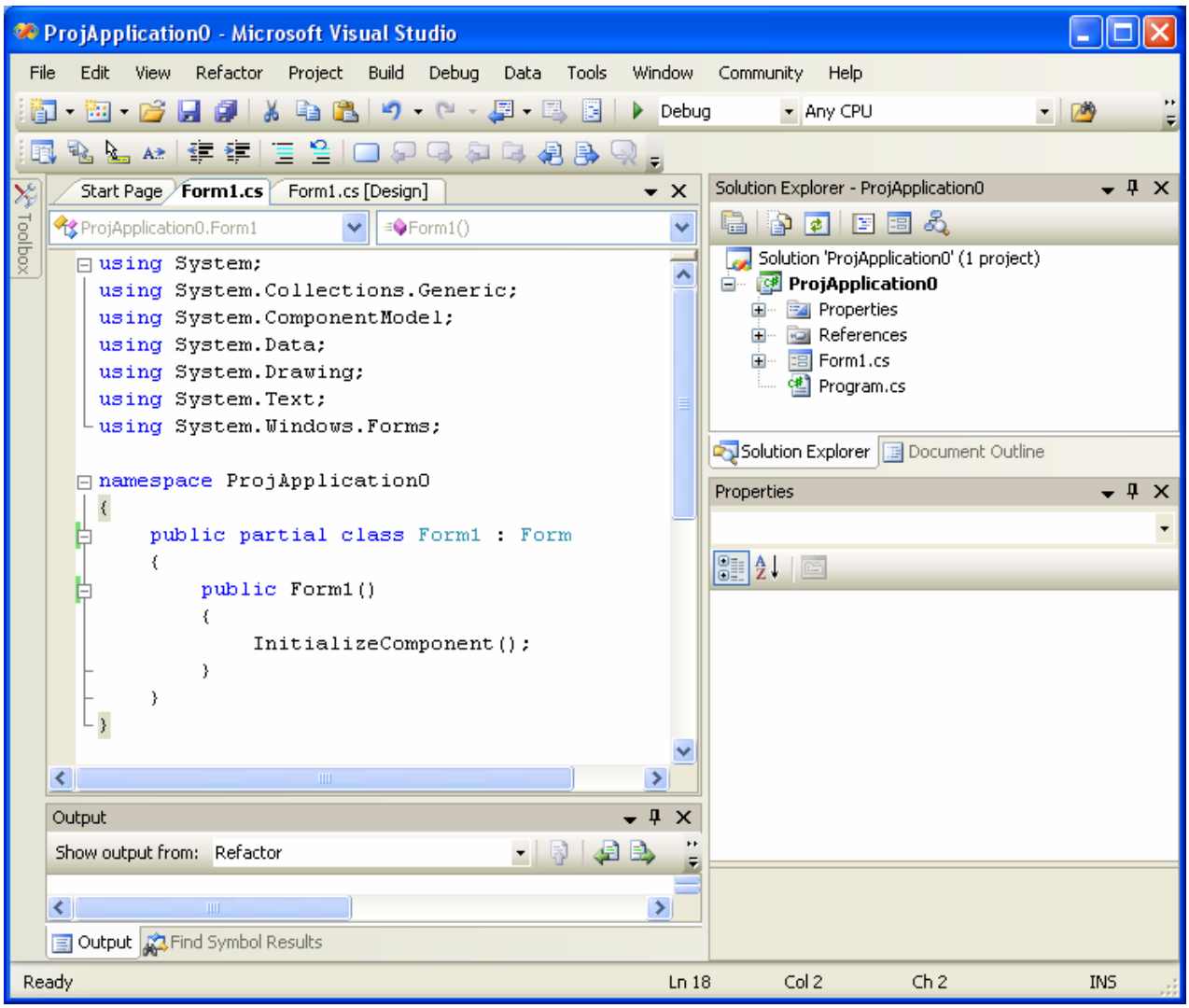
2°) Par exemple sélectionnons dans l'inspecteur d'objet de Visuel C#, la propriété FormBorderStyle *(le style par défaut est* FormBorderStyle.Sizable *)* modifions la à la valeur **None** et regardons dans l'onglet conception la nouvelle forme de la fiche :



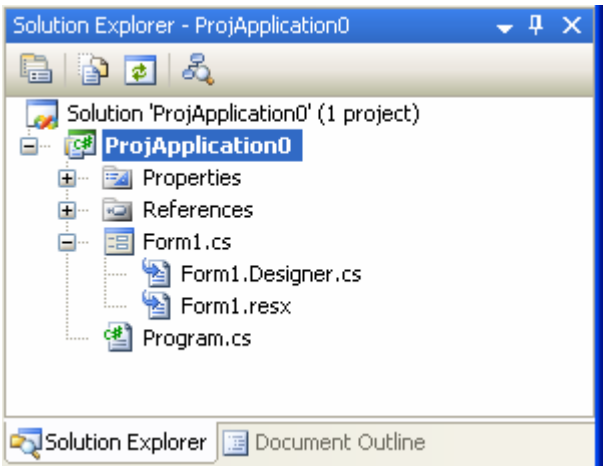
L'aspect visuel du formulaire a changé.

1.4 code C# engendré par le RAD pour un formulaire

Après avoir remis grâce à l'inspecteur d'objet, la propriété FormBorderStyle à sa valeur Sizable et remis le Name à la valeur initiale Form1, voyons maintenant en supposant avoir appelé notre application **ProjApplication0** ce que Visual C# a engendré comme code source que nous trouvons dans l'onglet de code Form1.cs pour notre formulaire :

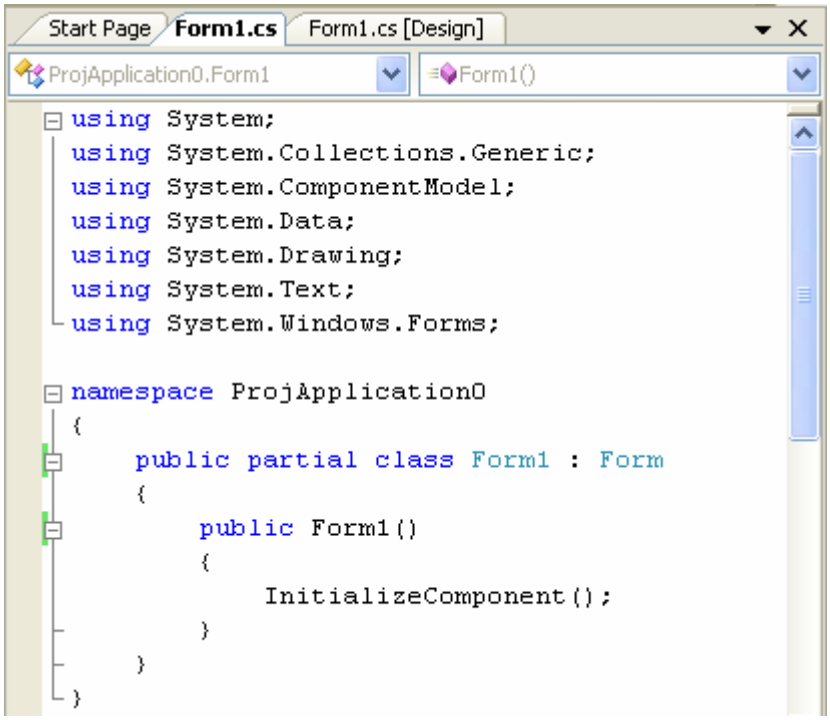


Le RAD Visuel C#, depuis la version 2.0, stocke les informations décrivant le programme dans 3 fichiers séparés (ici : Form1.cs, Form1.Designer.cs, Program.cs) :



La description de la **classe Form1** associée à la fiche est répartie dans les deux fichiers **Form1.cs** et **Form1.Designer.cs**.

Le fichier **Form1.cs** est celui qui contient les codes des gestionnaires d'événements que le développeur écrit au cours du codage :



Au départ ce fichier ne contient que le constructeur de la classe Form1: public Form1().

Ce constructeur fait appel à la méthode "private void InitializeComponent()" dont le code est situé dans le fichier **Form1.Designer.cs** et qui a pour but de décrire les composants à visualiser avec la fiche Form1et la fiche Form1elle-même.

Le fichier **Form1.Designer.cs** est celui qui contient les codes générés automatiquement par Visual C# pour décrire , la fiche, les composants, les propriétés, les delegate pour lier aux composants les gestionnaires d'événements qui sont écrits par le développeur dans le fichier **Form1.cs.** Enfin, ce

fichier contient aussi le code servant à permettre le nettoyage des resources utilisées, s'il elles existent.

```
Program.cs | Form1.Designer.cs | Start Page | Form1.cs | Form1.cs [Design]
ProjApplication0.Form1 | InitializeComponent()

namespace ProjApplication0
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should b
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
```

En "recollant" artificiellement à titre d'illustration, les contenus respectifs des deux fichiers **Form1.Designer.cs** et **Form1.cs** décrivant la classe Form1, on obtient en fait la description complète de cette classe :

```
class Form1
{
    public Form1()
    {
        InitializeComponent();
    }

    private System.ComponentModel.IContainer components = null;
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }
    private void InitializeComponent()
    {
```

Provient de Form1.cs

Provient de Form1.Designer.cs

```
        this.SuspendLayout();
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(292, 266);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
    }
}// fin de la classe Form1
```

Provient de Form1.Designer.cs

Le troisième fichier utilisé par le RAD se nomme **Program.cs**, il contient le code de la classe "principale" nommée Program, permettant le lancement de l'exécution du programme lui-même :

Program.cs | Form1.Designer.cs | Start Page | Form1.cs | Form1.cs [Design]

ProjApplication0.Program | Main()

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace ProjApplication0
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Quelques éléments explicatifs de l'analyse du code :

La méthode	correspond
static void Main( ) **{** Application.EnableVisualStyles(); Application.SetCompatibleTextRenderingDefault(false); Application.Run (**new** Form1 ( )); **}**	Point d'entrée d'exécution de l'application, son corps contient un appel de la méthode statique Run de la classe Application, elle instancie un objet "**new** Form1 ( )"de classe Form1 passé en paramètre à la méthode Run : c'est la fiche principale de l'application.

Application.Run (**new** Form1 ( )); La classe Application (semblable à TApplication de Delphi) fournit des membres statiques (propriétés et méthodes de classes) pour gérer une application (démarrer, arrêter une application, traiter des messages Windows), ou d'obtenir des informations sur une application. Cette classe est **sealed** et ne peut donc pas être héritée.	
La méthode Run de la classe Application dont voici la signature : **public static void** Run(ApplicationContext context);	Exécute une boucle de messages d'application standard sur le thread en cours, par défaut le paramètre context écoute l'événement **Closed** sur la fiche principale de l'application et dès lors arrête l'application.

Pour les connaisseurs de Delphi , le démarrage de l'exécution s'effectue dans le programme principal :

```
program Project1;
uses   Forms,  Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm (WinForm , Form1);
  Application.Run;
end.
```

Pour les connaisseurs des Awt et des Swing de Java, cette action C# correspond aux lignes suivantes :

Java2 avec Awt

```
class WinForm extends Frame {
  public WinForm ( ) {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  }
  protected void processWindowEvent(WindowEvent e) {
      super.processWindowEvent(e);
      if(e.getID( ) == WindowEvent.WINDOW_CLOSING) {
        System.exit(0); }
  }
  public static void main(String[] x )      {
          new WinForm ( );
  }
}
```

Java2 avec Swing

```
class WinForm extends JFrame {
  public WinForm ( ) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  public static void main(String[] x )      {
          new WinForm ( );
  }
}
```

Lorsque l'on regarde de plus près le code de la classe **Form1** situé dans **Form1.Designer.cs** on se rend compte qu'il existe une ligne en grisé située après la méthode **Dispose** :

```
namespace ProjApplication0
{
   partial class Form1
   {
      private System.ComponentModel.IContainer components = null;
      protected override void Dispose(bool disposing)
      {
         if (disposing && (components != null))
         {
            components.Dispose();
         }
         base.Dispose(disposing);
      }
   Windows Form Designer generated code
   }
}
```

Provient de Form1.Designer.cs

Il s'agit en fait de code replié (masqué) et généré automatiquement par le RAD, il est déconseillé de le modifier. Si nous le déplions et nous voyons apparaître la méthode privée InitializeComponent( ) contenant du code qui a manifestement été généré directement. En outre cette méthode est appelée dans le constructeur d'objet Form1 :

Voici le contenu exact de l'onglet code avec sa zone de code replié :

```
#region Windows Form Designer generated code
         /// <summary>
         /// Required method for Designer support - do not modify
         /// the contents of this method with the code editor.
         /// </summary>
         private void InitializeComponent()
         {
            this.SuspendLayout();
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(292, 266);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
         }
#endregion
```

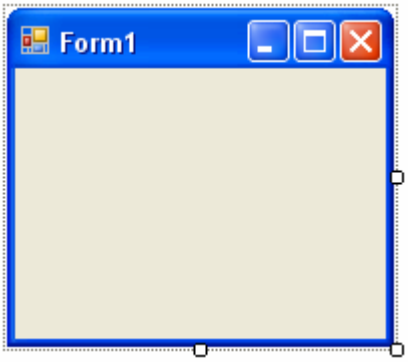
Lors du lancement du programme, la méthode Main() appelle le constructeur Form1( ) :

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

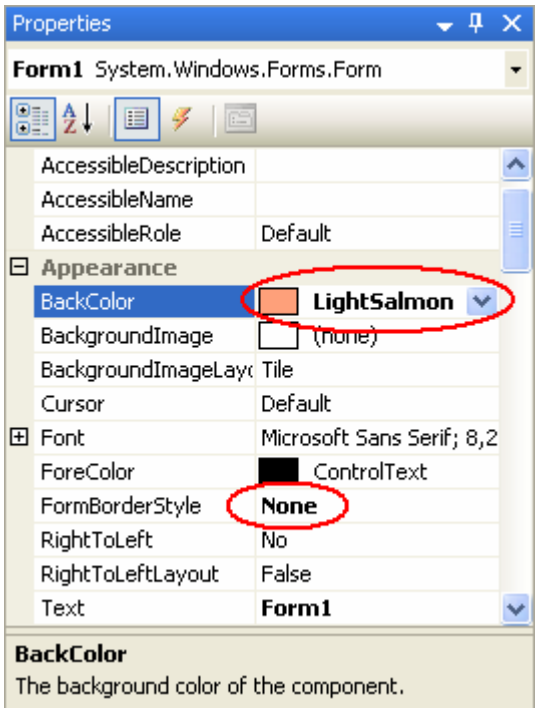
le constructeur Form1( ) appelle la méthode InitializeComponent() :

```
class Form1
{
    public Form1()
    {
      InitializeComponent();
    }
    private System.ComponentModel.IContainer components = null;
    protected override void Dispose(bool disposing)
    {
      if (disposing && (components != null))
      {
        components.Dispose();
      }
      base.Dispose(disposing);
    }
    private void InitializeComponent()
    {
      this.SuspendLayout();
      //
      // Form1
      //
      this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
      this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
      this.ClientSize = new System.Drawing.Size(292, 266);
      this.Name = "Form1";
      this.Text = "Form1";
      this.ResumeLayout(false);
    }
  }
```

La fiche Form1 est alors affichée sur l'écran :



Essayons de voir comment une manipulation visuelle engendre des lignes de code, pour cela modifions dans l'inspecteur d'objet deux propriétés FormBorderStyle et BackColor, la première est mise à None la seconde qui indique la couleur du fond de la fiche est mise à LightSalmon :



Consultons après cette opération le contenu du nouveau code généré, nous trouvons deux nouvelles lignes de code correspondant aux nouvelles actions visuelles effectuées (les nouvelles lignes sont figurées en rouge) :

```
#region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.BackColor = System.Drawing.Color.LightSalmon;
            this.ClientSize = new System.Drawing.Size(292, 266);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.None;
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
        }
#endregion
```

1.5 Libération de ressources non managées

Dans le code engendré par Visual studio ou C# Builder, nous avons laissé de côté la méthode Dispose :

```
protected override void Dispose (bool disposing)

{
        if (disposing && (components != null) )  {
              components.Dispose( );
        }
        base.Dispose( disposing );
 }
```

Pour comprendre son utilité, il nous faut avoir quelques lumières sur la façon que NetFrameWork a de gérer les ressources, rappelons que le CLR exécute et gére le code administré c'est à dire qu'il vérifie la validité de chaque action avant de l'exécuter. Le code non administré ou ressource non managée en C# est essentiellement du code sur les pointeurs qui doivent être déclarés **unsafe** pour pouvoir être utilisés, ou bien du code sur des fichiers, des flux , des handles .

La méthode **Dispose** existe déjà dans la classe mère **System.ComponentModel.Component** sous forme de deux surcharges avec deux signatures différentes. Elle peut être utile si vous avez mobilisé des ressources personnelles (on dit aussi ressources non managées) et que vous souhaitiez que celles-ci soient libérées lors de la fermeture de la fiche :

classe : **System.ComponentModel.Component**

méthode : **public virtual void** Dispose( );
Libère toutes les ressources utilisées par Component.

méthode : **protected virtual void** Dispose(bool disposing);
Libère uniquement les ressources non managées utilisées par Component..
ou
Libère toutes les ressources utilisées par Component.

Selon la valeur de disposing

- disposing = **true** pour libérer toutes les ressources (managées et non managées) ;
- disposing = **false** pour libérer uniquement les ressources non managées.

Remarque-1

Notons que pour le débutant cette méthode ne sera jamais utilisée et peut être omise puisqu'il s'agit d'une surcharge dynamique de la méthode de la classe mère.

Remarque-2

Il est recommandé par Microsoft, qu'un objet Component libère des ressources explicitement en appelant sa méthode Dispose **sans attendre une gestion automatique de la mémoire lors d'un appel implicite au Garbage Collector.**

Si nous voulons comprendre comment fonctionne le code engendré pour la méthode **Dispose**, il nous faut revenir à des éléments de base de la gestion mémoire en particulier relativement à la libération des ressources par le ramasse-miettes (garbage collector).

1.6 Comment la libération a-t-elle lieu dans le NetFrameWork ?

La classe mère de la hiérarchie dans le **NetFrameWork** est la classe **System.Object**, elle possède une méthode virtuelle **protected** Finalize, qui permet de libérer des ressources et d'exécuter d'autres opérations de nettoyage avant que Object soit récupéré par le garbage collecteur GC.

Bibliothèque de classes .NET Framework
Object, membres

Méthodes protégées

Finalize Pris en charge par le .NET Compact Framework.	Substitué. Autorise **Object** à tenter de libérer des ressources et d'exécuter d'autres opérations de nettoyage avant que **Object** soit récupéré par l'opération garbage collection. En C# et C++, les finaliseurs sont exprimés à l'aide de la syntaxe des destructeurs.

Lorsqu'un objet devient inaccessible il est automatiquement placé dans la **file d'attente de finalisation** de type FIFO, le garbage collecteur GC, lorsque la mémoire devient trop basse, effectue son travail en parcourant cette file d'attente de finalisation et en libérant la mémoire occupée par les objets de la file par appel à la méthode Finalize de chaque objet.

Donc si l'on souhaite libérer des ressources personnalisées, il suffit de redéfinir dans une classe fille la méthode Finalize( ) et de programmer dans le corps de la méthode la libération de ces ressources.

En C# on pourrait écrire pour une classe MaClasse :

```
protected override void Finalize( ) {
  try  {
    // libération des ressources personnelles
  }
  finally
  {
    base.Finalize( ); // libération des ressources du parent
  }
}
```

Mais syntaxiquement en C# la méthode Finalize n'existe pas et le code précédent, s'il représente bien ce qu'il faut faire, ne sera pas accepté par le compilateur. En C# la méthode Finalize s'écrit comme un destructeur de la classe MaClasse :

```
~MaClasse( ) {
 // libération des ressources personnelles
}
```

1.7 Peut-on influer sur cette la libération dans le NetFrameWork ?

Le processus de gestion de la libération mémoire et de sa récupération est entièrement automatisé dans le CLR, mais selon les nécessités on peut avoir le besoin de gérer cette désallocation : il existe pour cela, une classe **System.GC** qui autorise le développeur à une certaine dose de contrôle du garbage collector.

Par exemple, vous pouvez empêcher explicitement la méthode Finalize d'un objet figurant dans la file d'attente de finalisation d'être appelée, (utilisation de la méthode : **public static void** SuppressFinalize(object obj);)

Bibliothèque de classes .NET Framework

GC, membres

Méthodes publiques

ReRegisterForFinalize Pris en charge par le .NET Compact Framework.	Demande à ce que le système appelle la méthode du finaliseur pour l'objet spécifié pour lequel SuppressFinalize a été précédemment appelé.
SuppressFinalize Pris en charge par le .NET Compact Framework.	Demande à ce que le système n'appelle pas la méthode du finaliseur pour l'objet spécifié.

Vous pouvez aussi obliger explicitement la méthode Finalize d'un objet figurant dans la file d'attente de finalisation mais contenant GC.SuppressFinalize(...) d'être appelée, (utilisation de la méthode : **public static void** ReRegisterForFinalize(object obj);).

Microsoft propose deux recommandations pour la libération des ressources :

Il est recommandé d'empêcher les utilisateurs de votre application d'appeler directement la méthode Finalize d'un objet en limitant sa portée à protected.
Il est vivement déconseillé d'appeler une méthode Finalize pour une autre classe que votre classe de base directement à partir du code de votre application. **Pour supprimer correctement des ressources non managées, il est recommandé d'implémenter une méthode Dispose ou Close publique qui exécute le code de nettoyage nécessaire pour l'objet.**

Microsoft propose des conseils pour écrire la méthode Dispose :

1- La méthode Dispose d'un type doit libérer toutes les ressources qu'il possède.
2- Elle doit également libérer toutes les ressources détenues par ses types de base en appelant la méthode Dispose de son type parent. La méthode Dispose du type parent doit libérer toutes les ressources qu'il possède et appeler à son tour la méthode Dispose de son type parent, propageant ainsi ce modèle dans la hiérarchie des

types de base.
3- Pour que les ressources soient toujours assurées d'être correctement nettoyées, une méthode Dispose doit pouvoir être appelée en toute sécurité à plusieurs reprises sans lever d'exception.
4- Une méthode Dispose doit appeler la méthode GC.SuppressFinalize de l'objet qu'elle supprime.
5- La méthode Dispose doit être à liaison statique

1.8 Design Pattern de libération des ressources non managées

Le NetFrameWork propose une interface IDisposable ne contenant qu'une seule méthode : Dispose

Bibliothèque de classes .NET Framework

IDisposable, membres

IDisposable, vue d'ensemble

Méthodes publiques

Dispose Pris en charge par le .NET Compact Framework.	Exécute les tâches définies par l'application associées à la libération ou à la redéfinition des ressources non managées.

Rappel

Il est recommandé par Microsoft, qu'un objet Component libère des ressources explicitement en appelant sa méthode Dispose **sans attendre une gestion automatique de la mémoire lors d'un appel implicite au Garbage Collector. C'est ainsi que fonctionnent tous les contrôles et les composants de NetFrameWork. Il est bon de suivre ce conseil car dans le modèle de conception fourni ci-après, la libération d'un composant fait libérér en cascade tous les éléments de la hiérarchie sans les mettre en liste de finalisation ce qui serait une perte de mémoire et de temps pour le GC.**

Design Pattern de libération dans la classe de base

Voici pour information, proposé par Microsoft, un modèle de conception (Design Pattern) d'une classe MaClasseMere implémentant la mise à disposition du mécanisme de libération des ressources identique au NetFrameWork :

```
public class MaClasseMere : IDisposable {
  private bool disposed = false;
  //  un exemple de ressource managée : un composant
  private Component Components = new Component( );
  //  ...éventuellement des ressources non managées (pointeurs...)

  public void Dispose( )  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }
  protected virtual void Dispose(bool disposing)   {
    if (!this.disposed) {
      if (disposing) {
        Components.Dispose( );  //  libère le composant
         //  libère les autres éventuelles ressources managées
      }
      //  libère les ressources non managées :
      //... votre code de libération
     disposed = true;
  }
 ~MaClasseMere ( )  { // finaliseur par défaut
    Dispose(false);
  }
}
```

Ce modèle n'est présenté que pour mémoire afin de bien comprendre le modèle pour une classe fille qui suit et qui correspond au code généré par le RAD C# .

Design Pattern de libération dans une classe fille

Voici proposé le modèle de conception simplifié (Design Pattern) d'une classe MaClasseFille descendante de MaClasseMere, la classe fille contient une ressource de type System.ComponentModel.Container

```
public class MaClasseFille  : MaClasseMere {
  private System.ComponentModel.Container components = null ;
  public MaClasseFille ( )   {
    // code du constructeur....
  }

  protected override void Dispose(bool disposing) {
   if (disposing)  {
       if (components != null) { // s'il y a réellemnt une ressource
              components.Dispose( ); // on Dispose cette ressource
        }
       // libération éventuelle d'autres ressources managées...
    }
   // libération des ressources personnelles (non managées)...
   base.Dispose(disposing); // on Dispose dans la classe parent
  }
}
```

Information NetFrameWork sur la classe Container :

System.ComponentModel.Container

La classe **Container** est l'implémentation par défaut pour l'interface IContainer, une instance s'appelle un conteneur.

Les conteneurs sont des objets qui encapsulent et effectuent le suivi de zéro ou plusieurs composants qui sont des objets visuels ou non de la classe **System.ComponentModel.Component**.

Les références des composants d'un conteneur sont rangées dans une file FIFO, qui définit également leur ordre dans le conteneur.

La classe **Container** suit le modèle de conception mentionné plus haut quant à la libération des ressources managées ou non. Elle possède deux surcharges de Dispose implémentées selon le Design Pattern : la méthode protected virtual void Dispose(bool disposing); et la méthode public void Dispose( ). Cette méthode libère toutes les ressources détenues par les objets managés stockés dans la FIFO du Container. Cette méthode appelle la méthode Dispose( ) de chaque objet référencé dans la FIFO.

Les formulaires implémentent IDisposable :

La classe **System.Windows.Forms.Form** hérite de la classe **System.ComponentModel.Component**, or si nous consultons la documentation technique nous constatons que la classe Component en autres spécifications implémente l'interface IDisposable :

public class Component : MarshalByRefObject, IComponent, IDisposable

Donc tous les composants de C# sont construits selon le Design Pattern de libération :

La classe Component implémente le Design Pattern de libération de la classe de base et toutes les classe descendantes dont la classe Form implémente le Design Pattern de libération de la classe fille.

Nous savons maintenant à quoi sert la méthode Dispose **dans le code engendré par le RAD, elle nous propose une libération automatique des ressources de la liste des composants que nous aurions éventuellement créés :**

```
// Nettoyage des ressources utilisées :
protected override void Dispose (bool disposing)
{
    if (disposing && (components != null) )  { // Nettoyage des ressources managées
            components.Dispose( );
     }
     // Nettoyage des ressources non managées
     base.Dispose(disposing);
}
```

1.9 Un exemple utilisant la méthode Dispose d'un formulaire

Supposons que nous avons construit un composant personnel dans une classe **UnComposant** qui hérite de la classe Component selon le Design Pattern précédent, et que nous avons défini cette classe dans le namespace ProjApplication0 :

System.ComponentModel.Component
|__ProjApplication0.**UnComposant**

Nous voulons construire une application qui est un formulaire et nous voulons créer lors de l'initialisation de la fiche un objet de classe **UnComposant** que notre formulaire utilisera.

A un moment donné notre application ne va plus se servir du tout de notre composant, si nous souhaitons gérer la libération de la mémoire allouée à ce composant, nous pouvons :

- Soit attendre qu'il soit éligible au GC, en ce cas la mémoire sera libérée lorsque le GC le décidera,
- Soit le recenser auprès du conteneur de composants **components** (l'ajouter dans la FIFO de **components** si le conteneur a déjà été créé). Sinon nous créons ce conteneur et nous utilisons la méthode d'ajout (Add) de la classe **System.ComponentModel.Container** pour ajouter notre objet de classe **UnComposant** dans la FIFO de l'objet conteneur **components**.

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

namespace ProjApplication0
{
   /// <summary>
    /// Description Résumé de Form1.
    /// </summary>
   public partial class Form1 : System.Windows.Forms.Form
   {
      /// <summary>
       /// Variable requise par le concepteur.
       /// </summary>
      private System.ComponentModel.Container components = null;
      private UnComposant MonComposant = new UnComposant( );

      public Form1 ( )
      {

         InitializeComponent( );

         if ( components == null )
            components = new Container( );
         components.Add ( MonComposant );
      }
      /// <summary>
       /// Nettoyage des ressources utilisées.
       /// </summary>
      protected override void Dispose (bool disposing)
      {
         if (disposing && (components != nul ) )  {
               components.Dispose( );
            //-- libération ici d'autres ressources managées...
         }
         //-- libération ici de vos ressources non managées...
         base.Dispose(disposing);
      }


      region Code généré par le concepteur Windows Form

public class P¨rogram
{
      /// <summary>
      /// Le point d'entrée principal de l'application.
      /// </summary>
      [STAThread]
      static void Main( )
      {
         …..
         Application.Run(new Form1 ( ));
      }
   }
}
```

Déclaration-instanciation d'un composant personnel.

Ajout du composant personnel dans la Fifo de components.

Notre composant personnel est libéré avec les autres

1.10 L'instruction using appelle Dispose( )

La documentation technique signale que deux utilisations principales du mot clé **using** sont possibles :

Directive **using** :
Crée un alias pour un espace de noms ou importe des types définis dans d'autres espaces de noms.
*Ex: **using** System.IO ; **using** System.Windows.Forms ; ...*

Instruction **using** :
Définit une portée au bout de laquelle un objet est supprimé.

C'est cette deuxième utilisation qui nous intéresse : l'instruction **using**

<instruction **using**> **::= using (** <identif. Objet> | <liste de Déclar & instanciation> **)** <bloc instruction>
<bloc instruction> **::= {** < suite d'instructions > **}**
<identif. Objet> **::=** *un identificateur d'un objet existant et instancié*
<liste de Déclar & instanciation> **::=** *une liste séparée par des virgules de déclaration et initialisation d'objets semblable à la partie initialisation d'une boucle for.*

Ce qui nous donne deux cas d'écriture de l'instruction **using** :

1° - sur un objet déjà instancié :

```
classeA Obj = new classeA( );
....
using ( Obj )
{
  // code quelconque....
 }
```

2° - sur un (des) objet(s) instancié(s) localement au **using** :

```
using ( classeB Obj1 = new classeB ( ), Obj2 = new classeB ( ), Obj3 = new classeB ( ) )
{
  // code quelconque....
 }
```

Le **using** lance la méthode Dispose :

Dans les deux cas, on utilise une instance (Obj de classeA) ou l'on crée des instances (Obj1, Obj2 et Obj3 de classeB) dans l'instruction **using** pour garantir que la méthode **Dispose** est appelée sur l'objet lorsque l'instruction using est quittée.

Les objets que l'on utilise ou que l'on crée doivent implémenter l'interface System.IDisposable. Dans les exemples précédents classeA et classeB doivent implémenter elles-même ou par héritage l'interface System.IDisposable.

Exemple
Soit un objet visuel button1de classe System.Windows.Forms.Button, c'est la classe mère Control de Button qui implémente l'interface System.IDisposable :

```
public class Control : IComponent, IDisposable, IParserAccessor, IDataBindingsAccessor
```

Soient les lignes de code suivantes où **this** est une fiche :

```
// ....
this.button1 = new System.Windows.Forms.Button ( );
using( button1) {
  // code quelconque....
 }
// suite du code ....
```

A la sortie de l'instruction **using** juste avant la poursuite de l'exécution de la suite du code, button1.Dispose( ) a été automatiquement appelée par le CLR (le contrôle a été détruit et les ressources utilisées ont été libérées immédiatement).

1.11 L'attribut [STAThread]

Nous terminons notre examen du code généré automatiquement par le RAD pour une application fenêtrée de base, en indiquant la signification de l'attribut (mot entre crochets [STAThread]) situé avant la méthode Main :

```
/// <summary>
/// Le point d'entrée principal de l'application.
/// </summary>
[STAThread]
static void Main( )
{
   …..
   Application.Run(new Form1 ( ));
}
```

Cet attribut placé ici devant la méthode **Main** qualifie la manière dont le CLR exécutera l'application, il signifie : **S**ingle **T**hread **A**partments.

Il s'agit d'un modèle de gestion mémoire où l'application et tous ses composants est gérée dans un seul thread ce qui évite des conflits de ressources avec d'autres threads. Le développeur n'a pas à s'assurer de la bonne gestion des éventuels conflits de ressources entre l'application et ses composants.

Si on omet cet attribut devant la méthode Main, le CLR choisi automatiquement **[MTAThread]** **M**ulti **T**hread **A**partments, modèle de mémoire dans lequel l'application et ses composants sont gérés par le CLR en plusieurs thread, le développeur doit alors s'assurer de la bonne gestion des éventuels conflits de ressources entre l'application et ses composants.

Sauf nécessité d'augmentation de la fluidité du code, il faut laisser (ou mettre en mode console) l'attribut **[STAThread]** :

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

Des contrôles dans les formulaires

Plan général:

1. Les contrôles et les fonds graphiques

1.1 Les composants en général
1.2 Les contrôles sur un formulaire
1.3 Influence de la propriété parent sur l'affichage visuel d'un contrôle
1.4 Des graphiques dans les formulaires avec le GDI+
1.5 Le dessin doit être persistant
1.6 Deux exemples de graphiques sur plusieurs contrôles
- méthode générale pour tout redessiner
- des dessins sur événements spécifiques

1. Les contrôles et les fonds graphiques

Nous renvoyons le lecteur à la documentation du constructeur pour la manipulation de l'interface du RAD qu'il aura choisi. Nous supposons que le lecteur a aquis une dextérité minimale dans cette manimulation visuelle (déposer des composants, utiliser l'inspecteur d'objet (ou inspecteur de propriétés), modidifer des propriétés, créer des événements. Dans la suite du document, nous portons notre attention sur le code des programmes et sur leur comportement.

Car il est essentiel que le **lecteur sache par lui-même écrire le code** qui sera généré automatiquement par un RAD, sous peine d'être prisonnier du RAD et de ne pas pouvoir intervenir sur ce code !

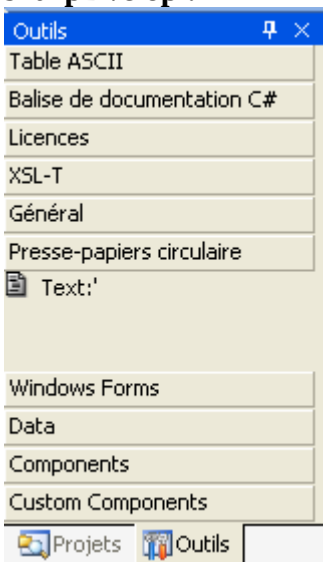
1.1 les composants en général

System.Windows.Forms.Control définit la classe de base des contrôles qui sont des composants avec représentation visuelle, les fiches en sont un exemple.

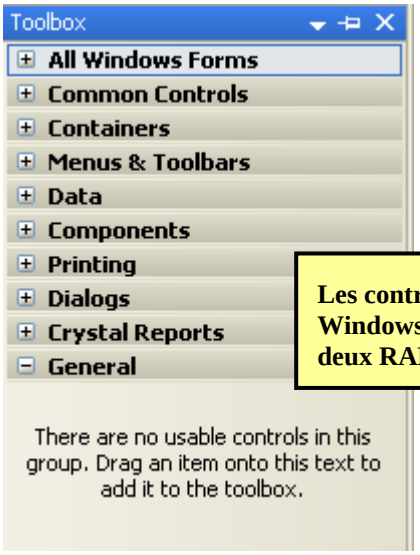
```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ContainerControl
            System.Windows.Forms.Form
```

Dans les RAD, la programmation visuelle des contrôles a lieu d'une façon très classique, par glisser déposer de composants situés dans une palette ou boîte d'outils. Il est possible de déposer visuellement des composants, certains sont visuels ils s'appellent contrôles, d'autres sont non visuels, ils s'appellent seulement composants.

sharpDvelop :

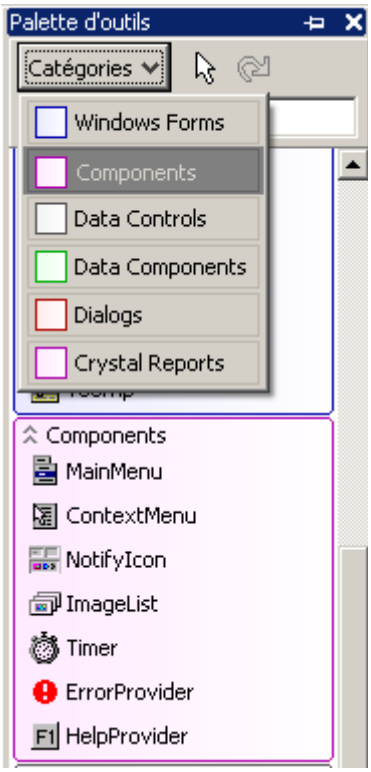


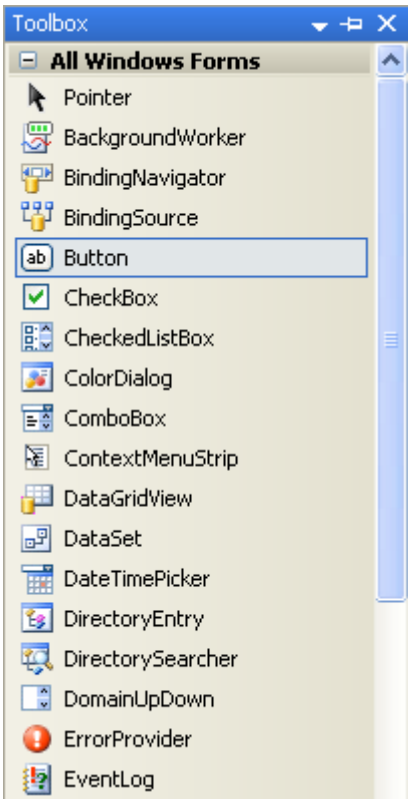
Visual C# :

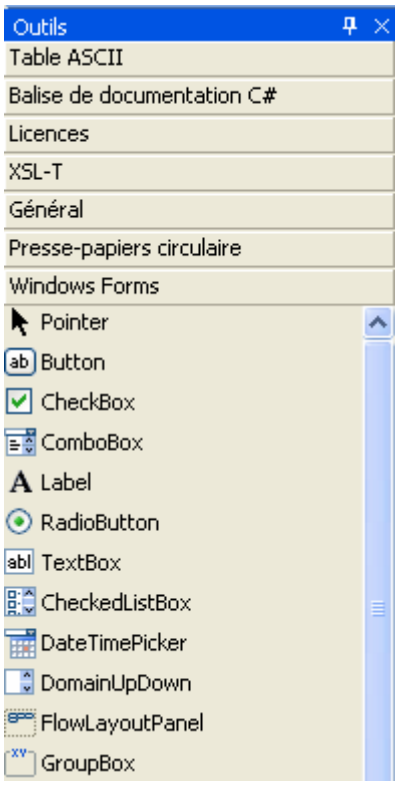


Les contrôles visuels ou Windows Forms dans deux RAD.

C# Builder **Visual C# :** **sharpdevelop :**



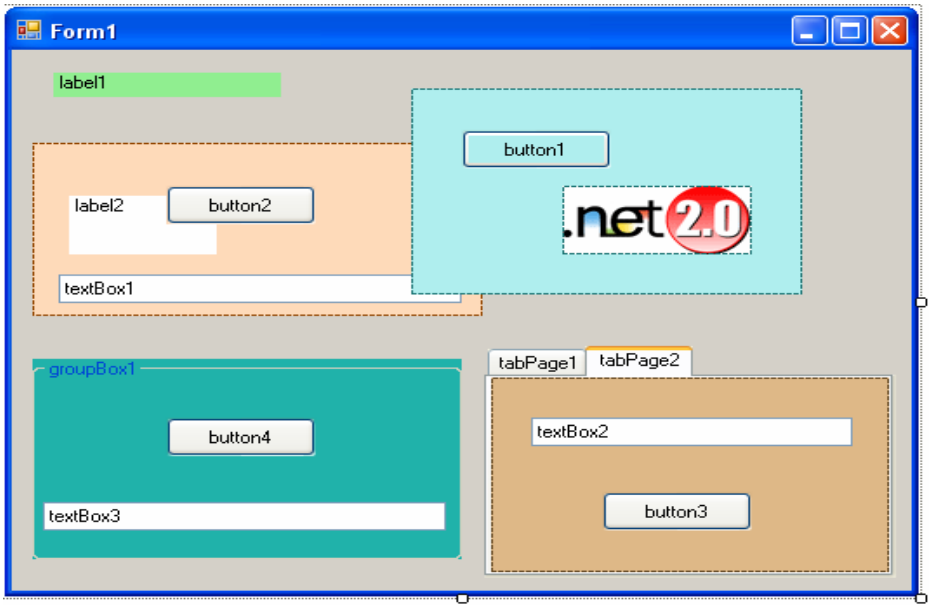




Pour pouvoir construire une IHM, il nous faut pouvoir utiliser à minima les composants visuels habituels que nous retrouvons dans les logiciels windows-like. Ici aussi la documentation technique fournie avec le RAD détaillera les différentes entités mises à disposition de l'utilisateur.

1.2 les contrôles sur un formulaire

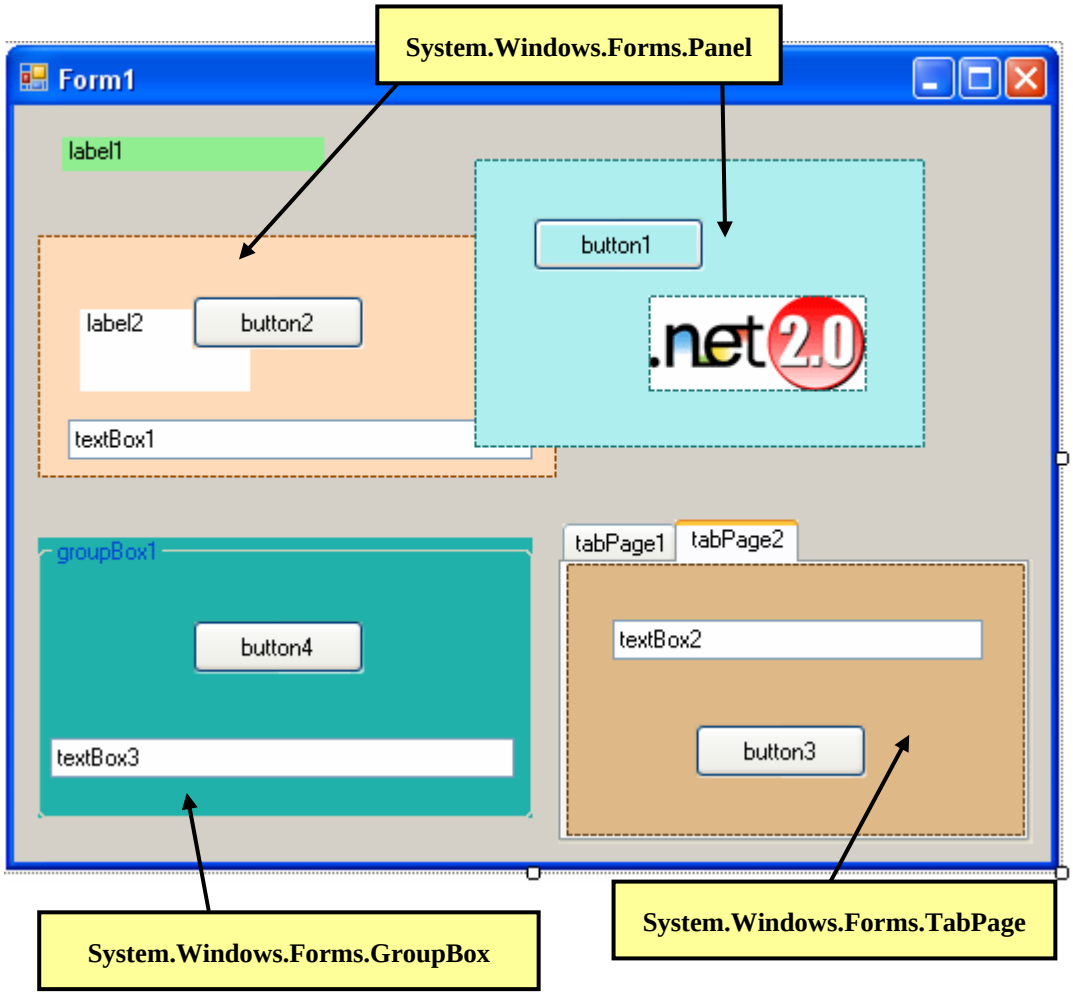
Voici un exemple de fiche comportant 7 catégories de contrôles différents :



Il existe des contrôles qui sont des conteneurs visuels, les quatre classes ci-après sont les principales classe de conteneurs visuels de C# :

System.Windows.Forms.Form
System.Windows.Forms.Panel
System.Windows.Forms.GroupBox
System.Windows.Forms.TabPage

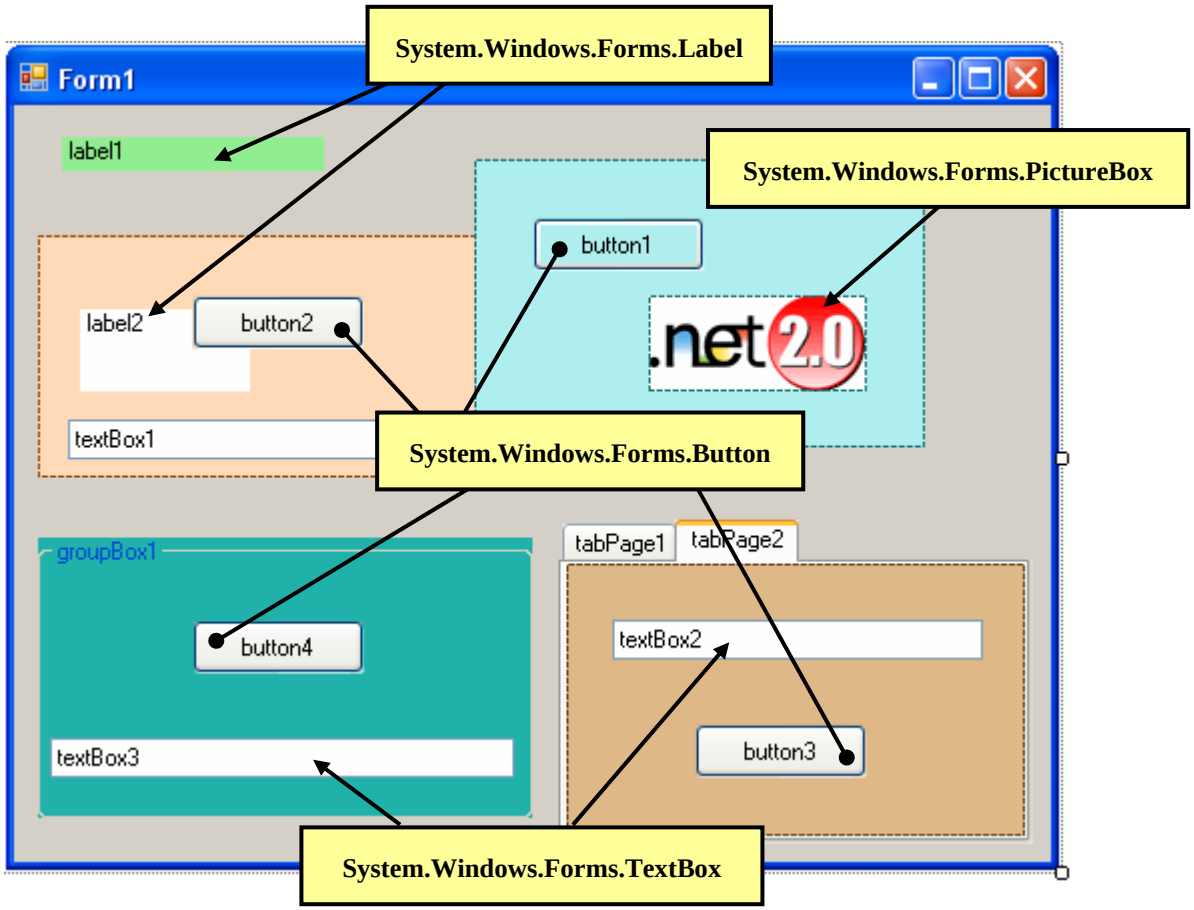
Sur la fiche précédente nous relevons outre le formulaire lui-même, quatre conteneurs visuels répartis en trois catégories de conteneurs visuels :



Un conteneur visuel permet à d'autres contrôles de s'afficher sur lui et lui communique par lien de parenté des valeurs de propriétés par défaut (police de caractères, couleur du fond,...). Un objet de chacune de ces classes de conteneurs visuels peut être le "parent" de n'importe quel contrôle grâce à sa propriété Parent qui est en lecture et écriture :

public Control Parent {**get**; **set**;}
C'est un objet de classe **Control** qui représente le conteneur visuel du contrôle.

Sur chaque conteneur visuel a été déposé un contrôle de classe **System.Windows.Forms.Button** qui a "hérité" par défaut des caractéristiques de police et de couleur de fond de son parent. Ci-dessous les classes de tous les contrôles déposés :

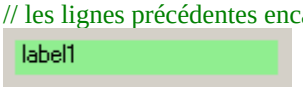


Le code C# engendré dans "Form1.Designer.cs" pour cette interface :

```
namespace ProjApplication0
{
   partial class Form1
   {
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
      protected override void Dispose(bool disposing)
      {
         if (disposing && (components != null))
         {
            components.Dispose();
         }
```

```
    base.Dispose(disposing);
}

#region Windows Form Designer generated code

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.panel1 = new System.Windows.Forms.Panel();
    this.button2 = new System.Windows.Forms.Button();
    this.label2 = new System.Windows.Forms.Label();
    this.button1 = new System.Windows.Forms.Button();
    this.panel2 = new System.Windows.Forms.Panel();
    this.pictureBox1 = new System.Windows.Forms.PictureBox();
    this.groupBox1 = new System.Windows.Forms.GroupBox();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.textBox3 = new System.Windows.Forms.TextBox();
    this.button4 = new System.Windows.Forms.Button();
    this.tabPage2 = new System.Windows.Forms.TabPage();
    this.tabPage1 = new System.Windows.Forms.TabPage();
    this.tabControl1 = new System.Windows.Forms.TabControl();
    this.button3 = new System.Windows.Forms.Button();
    this.textBox2 = new System.Windows.Forms.TextBox();
    this.panel1.SuspendLayout();
    this.panel2.SuspendLayout();
    ((System.ComponentModel.ISupportInitialize)(this.pictureBox1)).BeginInit();
    this.groupBox1.SuspendLayout();
    this.tabPage2.SuspendLayout();
    this.tabControl1.SuspendLayout();
    this.SuspendLayout();
    //
    // label1
    //
    this.label1.BackColor = System.Drawing.Color.LightGreen;
    this.label1.Location = new System.Drawing.Point(24, 16);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(131, 17);
    this.label1.TabIndex = 0;
    this.label1.Text = "label1";
```

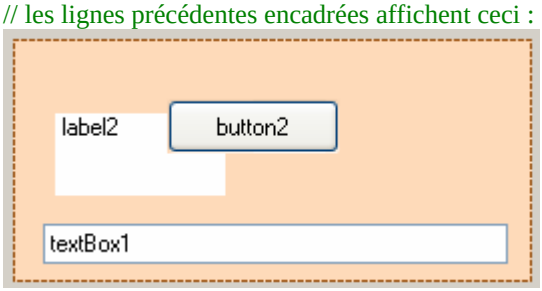
// les lignes précédentes encadrées affichent ceci :




```
    //
    // panel1
    //
    this.panel1.BackColor = System.Drawing.Color.PeachPuff;
    this.panel1.Controls.Add(this.textBox1);
    this.panel1.Controls.Add(this.button2);
    this.panel1.Controls.Add(this.label2);
    this.panel1.Location = new System.Drawing.Point(12, 65);
    this.panel1.Name = "panel1";
    this.panel1.Size = new System.Drawing.Size(259, 121);
    this.panel1.TabIndex = 1;
    //
    // button2
    //
```

```
this.button2.Location = new System.Drawing.Point(77, 30);
this.button2.Name = "button2";
this.button2.Size = new System.Drawing.Size(86, 27);
this.button2.TabIndex = 1;
this.button2.Text = "button2";
this.button2.UseVisualStyleBackColor = true;
//
// label2
//
this.label2.BackColor = System.Drawing.Color.White;
this.label2.Location = new System.Drawing.Point(21, 37);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(85, 41);
this.label2.TabIndex = 0;
this.label2.Text = "label2";
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(16, 76);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(232, 20);
this.textBox1.TabIndex = 2;
this.textBox1.Text = "textBox1";
```

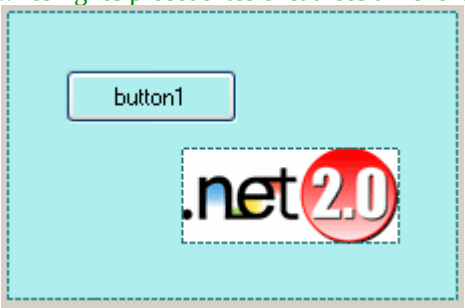
// les lignes précédentes encadrées affichent ceci :




```
//
// button1
//
this.button1.BackColor = System.Drawing.Color.PaleTurquoise;
this.button1.Location = new System.Drawing.Point(29, 29);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(86, 27);
this.button1.TabIndex = 1;
this.button1.Text = "button1";
this.button1.UseVisualStyleBackColor = false;
//
// panel2
//
this.panel2.BackColor = System.Drawing.Color.PaleTurquoise;
this.panel2.Controls.Add(this.pictureBox1);
this.panel2.Controls.Add(this.button1);
this.panel2.Location = new System.Drawing.Point(230, 27);
this.panel2.Name = "panel2";
this.panel2.Size = new System.Drawing.Size(225, 144);
this.panel2.TabIndex = 2;
//
// pictureBox1
//
this.pictureBox1.Image = global::ProjApplication0.Properties.Resources.net20;
this.pictureBox1.Location = new System.Drawing.Point(87, 68);
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size(109, 48);
```

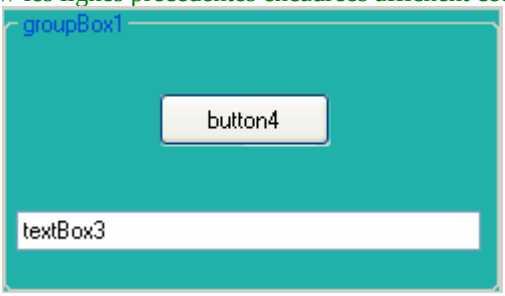
```
this.pictureBox1.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize;
this.pictureBox1.TabIndex = 2;
this.pictureBox1.TabStop = false;
```

// les lignes précédentes encadrées affichent ceci :




```
//
// groupBox1
//
this.groupBox1.BackColor = System.Drawing.Color.LightSeaGreen;
this.groupBox1.Controls.Add(this.textBox3);
this.groupBox1.Controls.Add(this.button4);
this.groupBox1.Location = new System.Drawing.Point(12, 216);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(247, 140);
this.groupBox1.TabIndex = 3;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "groupBox1";
//
// textBox3
//
this.textBox3.Location = new System.Drawing.Point(6, 100);
this.textBox3.Name = "textBox3";
this.textBox3.Size = new System.Drawing.Size(232, 20);
this.textBox3.TabIndex = 4;
this.textBox3.Text = "textBox3";
//
// button4
//
this.button4.Location = new System.Drawing.Point(77, 41);
this.button4.Name = "button4";
this.button4.Size = new System.Drawing.Size(86, 27);
this.button4.TabIndex = 5;
this.button4.Text = "button4";
this.button4.UseVisualStyleBackColor = true;
```

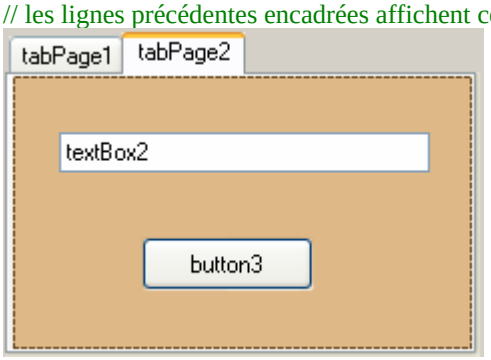
// les lignes précédentes encadrées affichent ceci :




```
//
// tabPage2
//
this.tabPage2.BackColor = System.Drawing.Color.BurlyWood;
this.tabPage2.Controls.Add(this.button3);
```

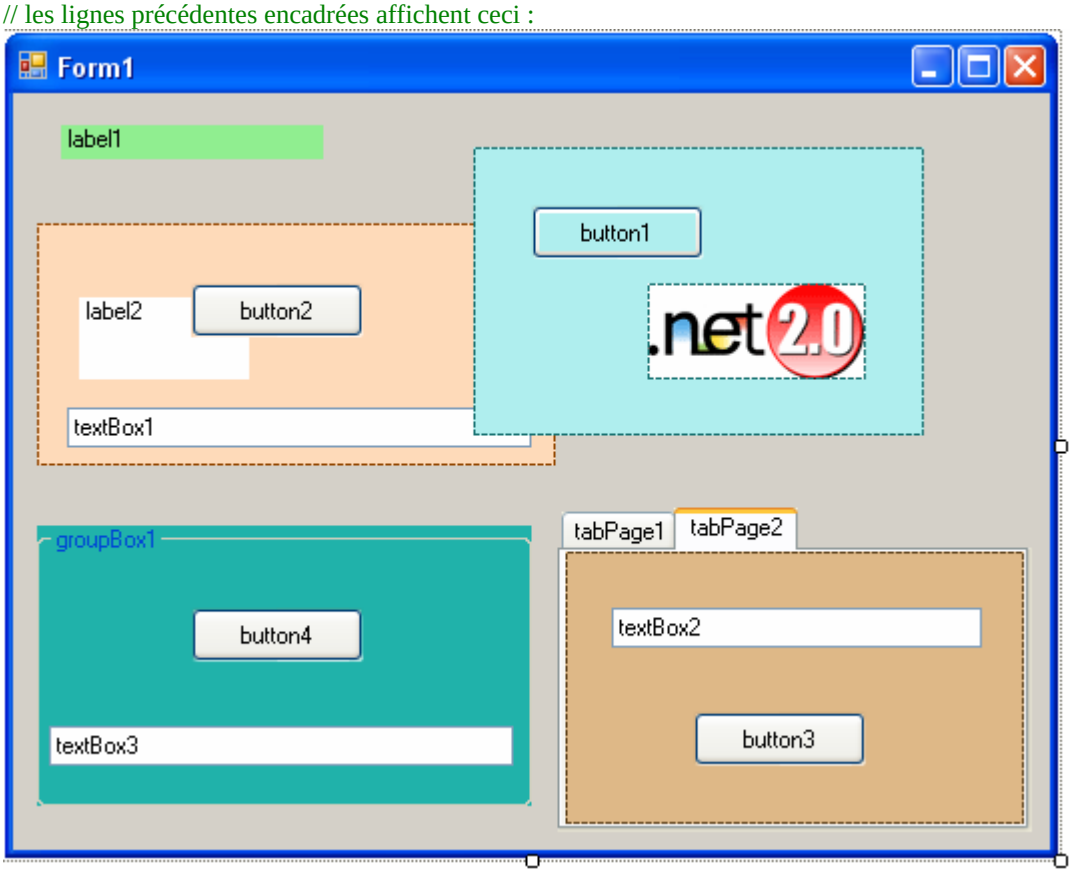
```
this.tabPage2.Controls.Add(this.textBox2);
this.tabPage2.Location = new System.Drawing.Point(4, 22);
this.tabPage2.Name = "tabPage2";
this.tabPage2.Padding = new System.Windows.Forms.Padding(3);
this.tabPage2.Size = new System.Drawing.Size(229, 136);
this.tabPage2.TabIndex = 1;
this.tabPage2.Text = "tabPage2";
//
// tabPage1
//
this.tabPage1.BackColor = System.Drawing.Color.MediumAquamarine;
this.tabPage1.Location = new System.Drawing.Point(4, 22);
this.tabPage1.Name = "tabPage1";
this.tabPage1.Padding = new System.Windows.Forms.Padding(3);
this.tabPage1.Size = new System.Drawing.Size(229, 136);
this.tabPage1.TabIndex = 0;
this.tabPage1.Text = "tabPage1";
this.tabPage1.UseVisualStyleBackColor = true;
//
// tabControl1
//
this.tabControl1.Controls.Add(this.tabPage1);
this.tabControl1.Controls.Add(this.tabPage2);
this.tabControl1.HotTrack = true;
this.tabControl1.Location = new System.Drawing.Point(272, 207);
this.tabControl1.Name = "tabControl1";
this.tabControl1.SelectedIndex = 0;
this.tabControl1.Size = new System.Drawing.Size(237, 162);
this.tabControl1.TabIndex = 4;
//
// button3
//
this.button3.Location = new System.Drawing.Point(64, 80);
this.button3.Name = "button3";
this.button3.Size = new System.Drawing.Size(86, 27);
this.button3.TabIndex = 4;
this.button3.Text = "button3";
this.button3.UseVisualStyleBackColor = true;
//
// textBox2
//
this.textBox2.Location = new System.Drawing.Point(23, 28);
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size(185, 20);
this.textBox2.TabIndex = 5;
this.textBox2.Text = "textBox2";
```

// les lignes précédentes encadrées affichent ceci :



```
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.BackColor = System.Drawing.SystemColors.ActiveBorder;
        this.ClientSize = new System.Drawing.Size(518, 378);
        this.Controls.Add(this.tabControl1);
        this.Controls.Add(this.groupBox1);
        this.Controls.Add(this.panel2);
        this.Controls.Add(this.panel1);
        this.Controls.Add(this.label1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.panel1.ResumeLayout(false);
        this.panel1.PerformLayout();
        this.panel2.ResumeLayout(false);
        this.panel2.PerformLayout();
        ((System.ComponentModel.ISupportInitialize)(this.pictureBox1)).EndInit();
        this.groupBox1.ResumeLayout(false);
        this.groupBox1.PerformLayout();
        this.tabPage2.ResumeLayout(false);
        this.tabPage2.PerformLayout();
        this.tabControl1.ResumeLayout(false);
        this.ResumeLayout(false);
```

// les lignes précédentes encadrées affichent ceci :




```
    }

    #endregion
```

```
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Panel panel1;
```

```
        private System.Windows.Forms.Button button2;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.Panel panel2;
        private System.Windows.Forms.PictureBox pictureBox1;
        private System.Windows.Forms.TextBox textBox1;
        private System.Windows.Forms.GroupBox groupBox1;
        private System.Windows.Forms.TextBox textBox3;
        private System.Windows.Forms.Button button4;
        private System.Windows.Forms.TabPage tabPage2;
        private System.Windows.Forms.Button button3;
        private System.Windows.Forms.TextBox textBox2;
        private System.Windows.Forms.TabPage tabPage1;
        private System.Windows.Forms.TabControl tabControl1;
    }
}
```

Déclaration des références des objets de type composants

1.3 Influence de la propriété parent sur l'affichage visuel d'un contrôle

Dans l'IHM précédente, programmons par exemple la modification de la propriété Parent du contrôle textBox1 en réaction au click de souris sur les Button button1 et button2.

Il faut abonner le gestionnaire du click de button1 "**private void** button1_Click" , au délégué button1.Click :

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

De même il faut abonner le gestionnaire du click de button2 "**private void** button2_Click" , au délégué button2.Click :

```
this.button2.Click += new System.EventHandler(this.button2_Click);
```

Le RAD engendre automatiquement les gestionnaires:

```
private void button1_Click ( object sender, System .EventArgs e ){ }
private void button1_Click ( object sender, System .EventArgs e ){ }
```

Les lignes d'abonnement sont engendrées dans la méthode InitializeComponent **( )** :

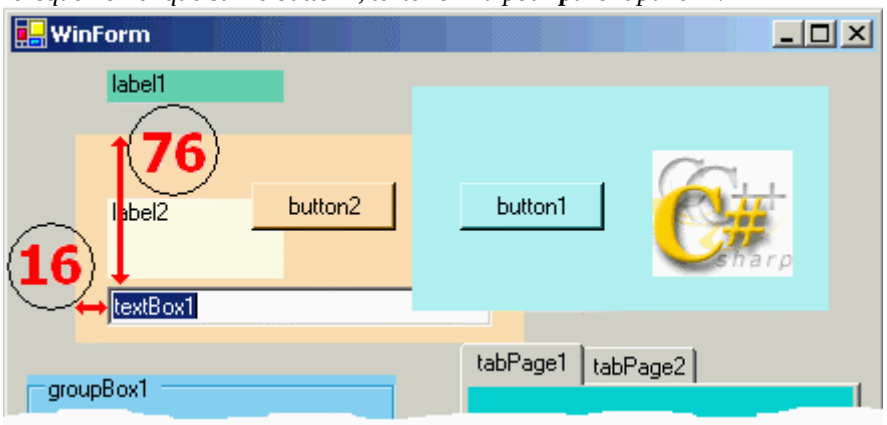
```
private void InitializeComponent ( )
{ ....

this.button1.Click += new System.EventHandler(this.button1_Click);
this.button2.Click += new System.EventHandler(this.button2_Click);
}
```

Le code et les affichages obtenus *(le textBox1 est positionné en X=16 et Y=76 sur son parent)* :

```
// Gestionnaire du click sur button1 :

private void button1_Click ( object sender, System .EventArgs e ){
  textBox1.Parent = panel1 ;
}
```

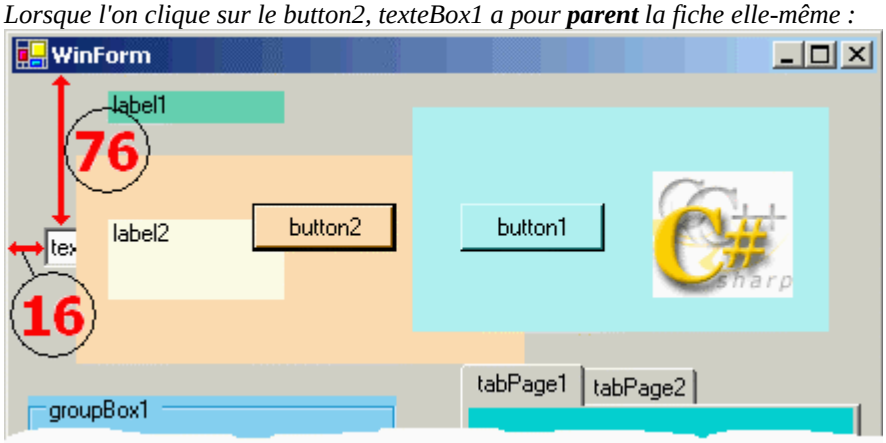
Le composant a déjà été positionné par l'instruction suivante :
this.textBox1.Location = new System.Drawing.Point(16, 76);

*Lorsque l'on clique sur le button1, texteBox1 a pour **parent** panel1 :*




```
// Gestionnaire du click sur button2 :

private void button2_Click ( object sender, System .EventArgs e ) {
  textBox1.Parent = this;
}
```

*Lorsque l'on clique sur le button2, texteBox1 a pour **parent** la fiche elle-même :*



Le contrôle pictureBox1 permet d'afficher des images : ico, bmp, gif, png, jpg, jpeg

```
// chargement d'un fichier image dans le pictureBox1 par un click sur button3 :

private void InitializeComponent ( )
{ ....
```

```
  this.button1.Click += new System.EventHandler(this.button1_Click);
  this.button2.Click += new System.EventHandler(this.button2_Click);
  this.button3.Click += new System.EventHandler(this.button3_Click);
}

private void  button3_Click ( object  sender,  System .EventArgs e ) {
   pictureBox1.Image  =  Image.FromFile("csharp.jpg") ;
}
```

Lorsque l'on clique sur le button3, l'image "csharp.jpg" est chargée dans pictureBox1 :

1.4 Des graphiques dans les formulaires avec le GDI+

Nous avons remarqué que C# possède un contrôle permettant l'affichage d'images de différents formats, qu'en est-il de l'affichage de graphiques construits pendant l'exécution ? Le GDI+ répond à cette question.

Le **Graphical Device Interface+** est la partie de NetFrameWork qui fournit les graphismes vectoriels à deux dimensions, les images et la typographie. GDI+ est une interface de périphérique graphique qui permet aux programmeurs d'écrire des applications indépendantes des périphériques physiques (écran, imprimante,...).

Lorsque l'on dessine avec GDI+, **on utilise des méthodes de classes situées dans le GDI+**, donnant des directives de dessin, ce sont ces méthodes qui, via le CLR du NetFrameWork, font appel aux pilotes du périphérique physique, **les programmes ainsi conçus ne dépendent alors pas du matériel sur lequel ils s'afficheront**.

Pour dessiner des graphiques sur n'importe quel périphérique d'affichage, il faut un objet **Graphics**. Un objet de classe **System.Drawing.Graphics** est associé à une surface de dessin, généralement la zone cliente d'un formulaire (objet de classe Form). Il n'y a pas de constructeur dans la classe Graphics :

Comme le dessin doit avoir lieu sur la surface visuelle d'un objet visuel donc un contrôle, c'est cet objet visuel qui fournit le fond, le GDI+ fournit dans la classe **System.Windows.Forms.Control**

la méthode **CreateGraphics** qui permet de créer un objet de type **Graphics** qui représente le "fond de dessin" du contrôle :

Bibliothèque de classes .NET Framework
Control, méthodes

CreateGraphics Pris en charge par le .NET Compact Framework.	Crée l'objet Graphics pour le contrôle.

Syntaxe :

> **public** Graphics CreateGraphics( );

Afin de comprendre comment utiliser un objet Graphics construisons un exemple fictif de code dans lequel on suppose avoir instancié ObjVisuel un contrôle (par exemple : une fiche, un panel,...), on utilise deux méthodes dessin de la classe Graphics pour dessiner un trait et un rectangle :

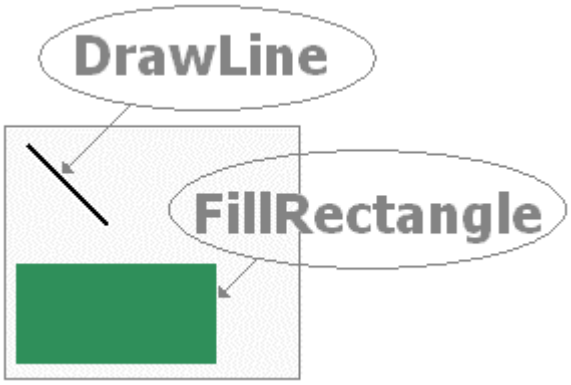
Bibliothèque de classes .NET Framework
Graphics, méthodes

DrawLine Pris en charge par le .NET Compact Framework.	Surchargé. Dessine une ligne reliant les deux points spécifiés par des paires de coordonnées.
FillRectangle Pris en charge par le .NET Compact Framework.	Surchargé. Remplit l'intérieur d'un rectangle spécifié par une paire de coordonnées, une largeur et une hauteur.

Code C#	Explication
Graphics fond = ObjVisuel.CreateGraphics **( );**	Obtention d'un fond de dessin sur ObjVisuel (création d'un objet Graphics associé à ObjVisuel)
Pen blackPen = **new** Pen **(** Color.Black, 2 **);**	Création d'un objet de pinceau de couleur noire et d'épaisseur 2 pixels
fond.DrawLine **(** blackPen, 10f, 10f, 50f, 50f **);**	Utilisation du pinceau blackPen pour tracer une ligne droite sur le fond d'ObjVisuel entre les deux points A(10,10) et B(50,50).
fond.FillRectangle **(** Brushes.SeaGreen,5,70,100,50 **);**	Utilisation d'une couleur de brosse SeaGreen, pour remplr l'intérieur d'un rectangle spécifié par une paire de coordonnées (5,70), une largeur(100 pixels) et une hauteur (50 pixels).

Ces quatre instructions ont permis de dessiner le trait noir et le rectangle vert sur le fond du

contrôle ObjVisuel représenté ci-dessous par un rectangle à fond blanc :



Note technique de Microsoft
L'objet Graphics retourné doit être supprimé par l'intermédiaire d'un appel à sa méthode ***Dispose*** *lorsqu'il n'est plus nécessaire.*

La classe Graphics implémente l'interface IDisposable :

public sealed class Graphics : MarshalByRefObject, IDisposable

Les objets Graphics peuvent donc être libérés par la méthode Dispose( ).

Afin de respecter ce conseil d'optimisation de gestion de la mémoire, nous rajoutons dans notre code l'appel à la méthode **Dispose** de l'objet Graphics. Nous prenons comme ObjVisuel un contrôle de type panel que nous nommons panelDessin (**private System.Windows.Forms.Panel** panelDessin):

```
//...
Graphics fond = panelDessin.CreateGraphics ( );
Pen blackPen = new Pen (  Color.Black, 2 );
fond.DrawLine (  blackPen, 10f, 10f, 50f, 50f );
fond.FillRectangle  (  Brushes.SeaGreen,5,70,100,50  );
fond.Dispose( );
//...suite du code où l'objet fond n'est plus utilisé
```

Nous pouvons aussi utiliser l'instruction **using** déjà vue qui libère automatiquement l'objet par appel à sa méthode Dispose :

```
//...
using( Graphics fond = panelDessin.CreateGraphics ( ) ) {
  Pen blackPen = new Pen (  Color.Black, 2 );
  fond.DrawLine (  blackPen, 10f, 10f, 50f, 50f );
  fond.FillRectangle  (  Brushes.SeaGreen,5,70,100,50  );
}
//...suite du code où l'objet fond n'est plus utilisé
```

1.5 Le dessin doit être persistant

Reprenons le dessin précédent et affichons-le à la demande.

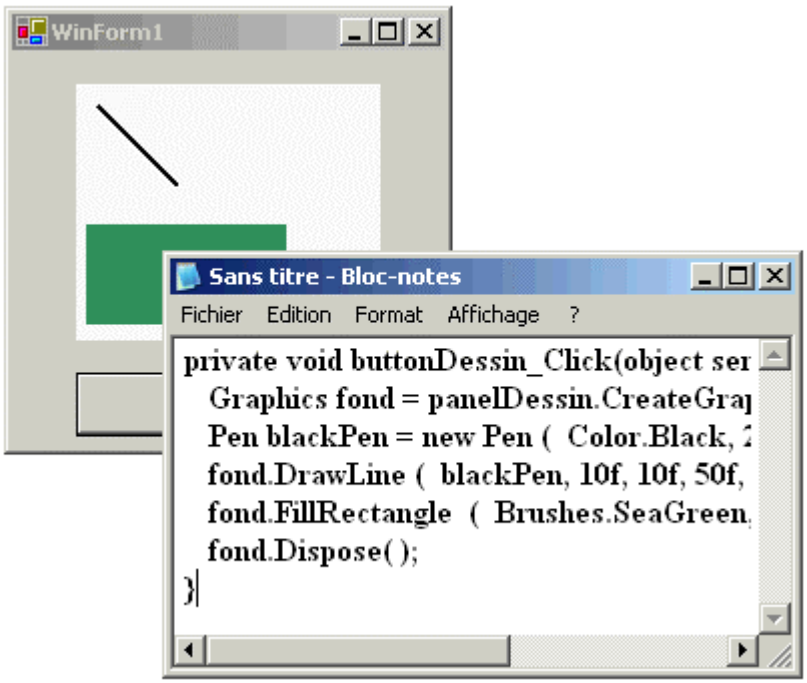
Nous supposons disposer d'un formulaire nommé WinForm1 contenant un panneau nommé panelDessin (**private System.Windows.Forms.Panel** panelDessin) et un bouton nommé buttonDessin (**private System.Windows.Forms.Button** buttonDessin)

Nous programmons un gestionnaire de l'événement click du buttonDessin, dans lequel nous copions le code de traçage de notre dessin :

```
private void buttonDessin_Click(object sender, System.EventArgs e)    {
   Graphics fond = panelDessin.CreateGraphics ( );
   Pen blackPen = new Pen (  Color.Black, 2 );
   fond.DrawLine (  blackPen, 10f, 10f, 50f, 50f );
   fond.FillRectangle  (  Brushes.SeaGreen,5,70,100,50  );
   fond.Dispose( );
}
```

Lorsque nous cliquons sur le bouton buttonDessin, le trait noir et le rectangle vert se dessine sur le fond du panelDessin :

Faisons apparaître une fenêtre de bloc-note contenant du texte, qui masque partiellement notre formulaire WinForm1 qui passe au second plan comme ci-dessous :



Si nous nous refocalisons sur le formulaire en cliquant sur lui par exemple, celui-ci repasse au premier plan, nous constatons que notre dessin est abîmé. Le rectangle vert est amputé de la partie qui était recouverte par la fenêtre de bloc-note. Le formulaire s'est bien redessiné, mais pas nos traçés ;

Il faut cliquer une nouvelle fois sur le bouton pour lancer le redessinement des tracés :

Il existe un moyen simple permettant d'effectuer le redessinement de nos traçés lorsque le formulaire se redessine lui-même automatiquement : il nous faut "consommer" l'événement **Paint** du formulaire qui se produit lorsque le formulaire est redessiné (ceci est d'ailleurs valable pour n'importe quel contrôle). La consommation de l'événement Paint s'effectue grâce au gestionnaire Paint de notre formulaire WinForm1 :

```
private void WinForm1_Paint (object sender, System.Windows.Forms.PaintEventArgs e)  {
  Graphics fond = panelDessin.CreateGraphics ( );
  Pen blackPen = new Pen (  Color.Black, 2 );
  fond.DrawLine (  blackPen, 10f, 10f, 50f, 50f );
  fond.FillRectangle  (  Brushes.SeaGreen,5,70,100,50  );
  fond.Dispose( );
}
```

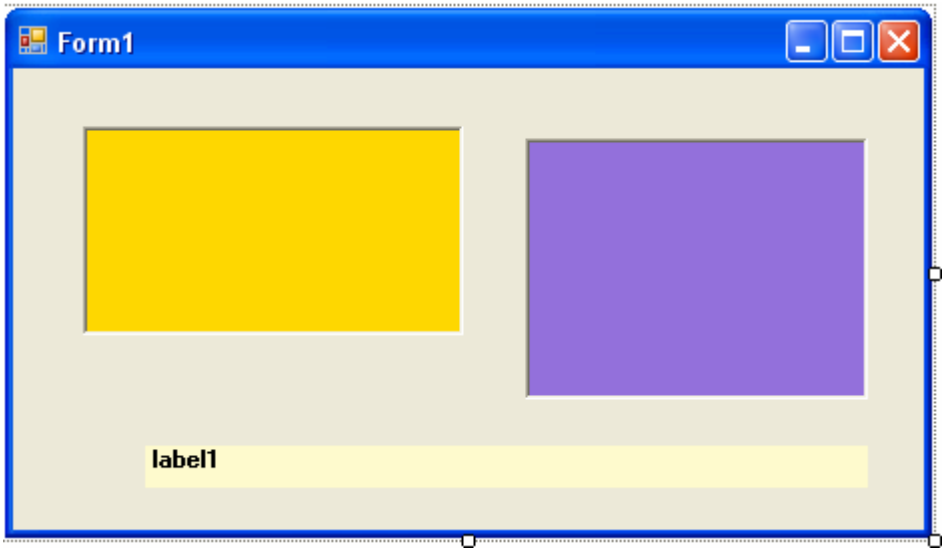
Le RAD a enregistré (abonné) le gestionnaire WinForm1_Paint auprès du délégué Paint dans le corps de la méthode InitializeComponent :

```
private void InitializeComponent( )  {
  ...
  this.Paint += new System.Windows.Forms.PaintEventHandler ( this.WinForm1_Paint );
  ...
}
```

1.6 Deux exemples de graphiques sur plusieurs contrôles

Premier exemple : une méthode générale pour tout redessiner.

Il est possible de dessiner sur tous les types de conteneurs visuels ci-dessous un formulaire nommé Form1 et deux panel (panel1 : jaune foncé et panel2 : violet), la label1 ne sert qu'à afficher du texte en sortie :



Nous écrivons une méthode **TracerDessin** permettant de dessiner sur le fond de la fiche, sur le fond des deux panel et d'écrire du texte sur le fond d'un panel. La méthode **TracerDessin** est appelée dans le gestionnaire de l'événement Paint du formulaire lors de son redessinement de la

fiche afin d'assurer la persistance de tous les traçés. Voici le code qui est créé :

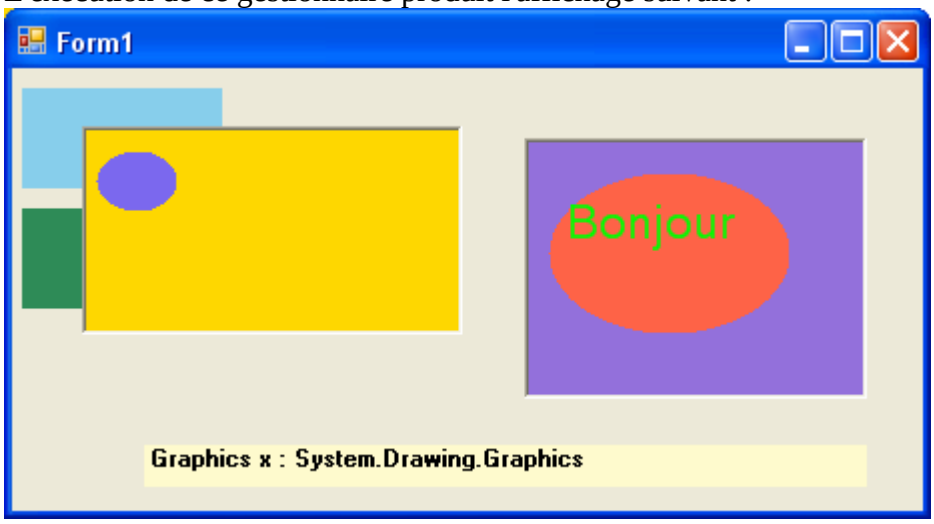
Automatiquement généré dans Form1.Designer.cs :

```
private void InitializeComponent( )  {
  ...
  this.Paint += new System.Windows.Forms.PaintEventHandler ( this Form1_Paint );
  ...
}
```

Ecrit dans le gestionnaire de l'événement Paint de Form1, dans Form1.cs :

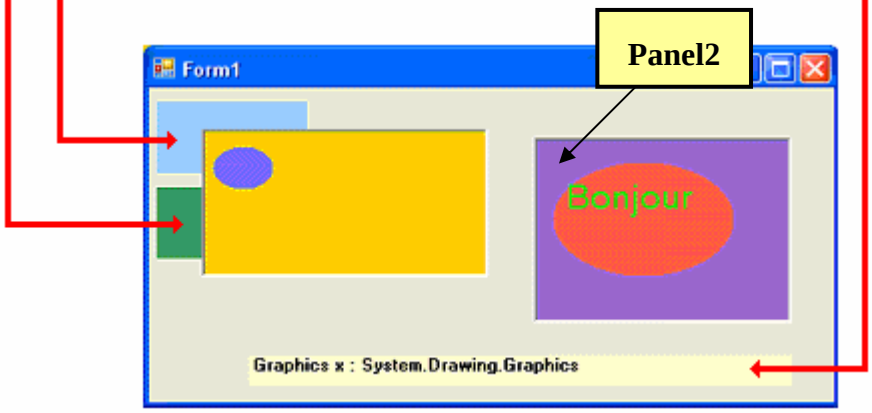
```
private void Form1_Paint (object sender, System.Windows.Forms.PaintEventArgs e)  {

        TracerDessin  ( e.Graphics  ) ;
/*  Explications sur l'appel de la méthode TracerDessin :
Le paramètre e de type PaintEventArgs contient les données relatives à l'événement Paint
en particulier une propriété Graphics qui renvoie le graphique utilisé pour peindre sur la fiche
c'est pourquoi e.Graphics est passé comme fond en paramètre à notre méthode de dessin.
*/
}

private void  TracerDessin ( Graphics x  ){
  string  Hdcontext ;
  Hdcontext =  x.GetType ( ) .ToString ( );
  label1.Text = "Graphics x : " + Hdcontext.ToString ( );
  x.FillRectangle ( Brushes.SeaGreen,5,70,100,50 );

 using( Graphics g  =  this .CreateGraphics ( ) )  {
   g.FillRectangle ( Brushes.SkyBlue,5,10,100,50 );
 }
using( Graphics g  =  panel1.CreateGraphics ( ) ) {
   g.FillEllipse ( Brushes.MediumSlateBlue,5,10,40,30 );
 }
using( Graphics h  =  panel2.CreateGraphics ( ) ) {
   h.FillEllipse ( Brushes.Tomato,10,15,120,80 );
   h.DrawString ("Bonjour" , new  Font (this .Font.FontFamily.Name,18 ) ,Brushes.Lime,15,25 );
 }
}
```

L'exécution de ce gestionnaire produit l'affichage suivant :



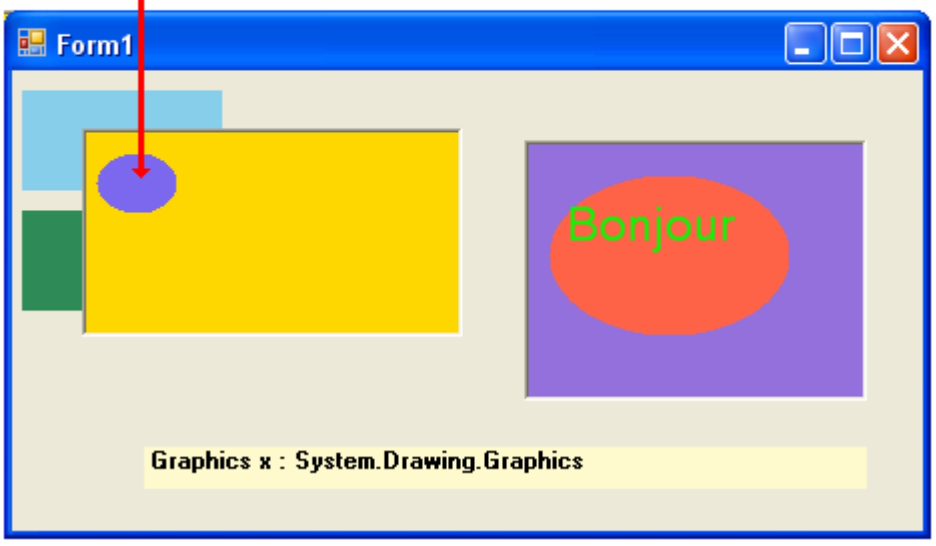
Illustrons les actions de dessin de chaque ligne de code de la méthode TracerDessin :

```
private void  TracerDessin ( Graphics x  ) {

    string Hdcontext ;
    Hdcontext =  x.GetType ( ) .ToString ( );
    label1 .Text = "Graphics x : " + Hdcontext.ToString ( );

    x.FillRectangle ( Brushes.SeaGreen,5,70,100,50 );

    using( Graphics g  =  this .CreateGraphics ( ) )  {
      g.FillRectangle ( Brushes.SkyBlue,5,10,100,50 );
    }
```

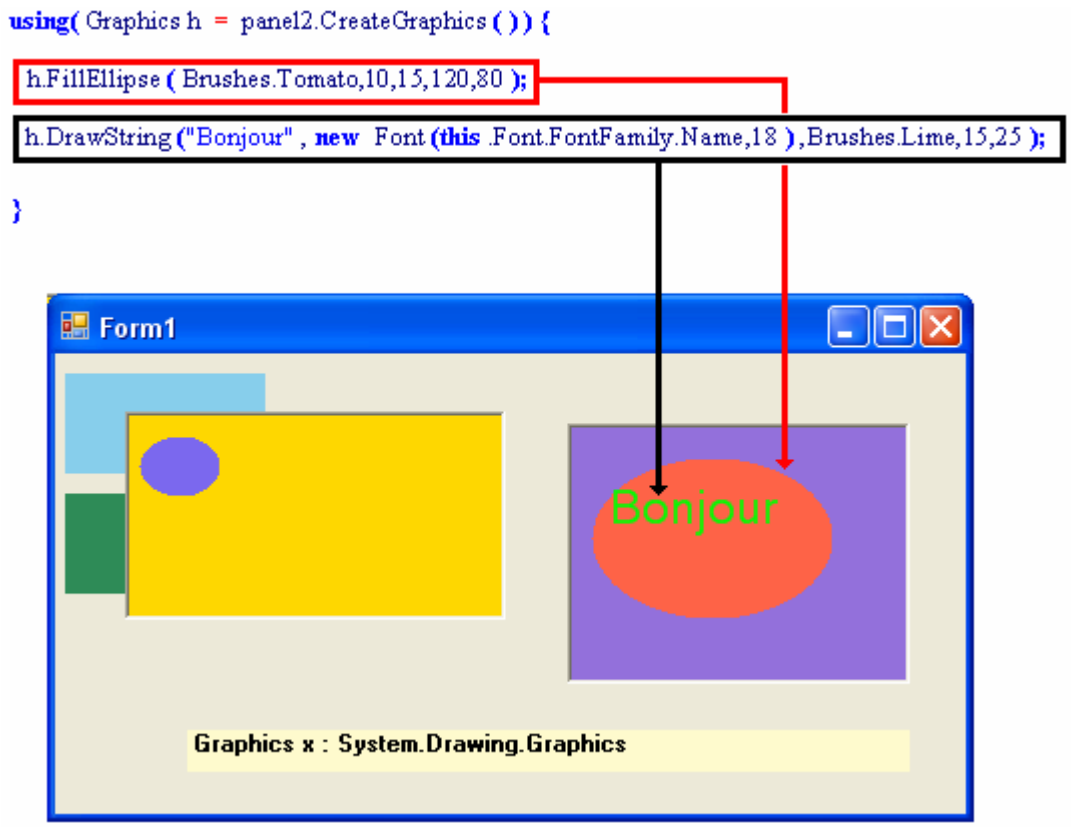


On dessine deux rectangles sur le fond de la fiche, nous notons que ces deux rectangles ont une intersection non vide avec le panel2 (jaune foncé)et que cela n'altère pas le dessin du panel. En effet le panel est un contrôle et donc se redessine lui-même. Nous en déduisons que le fond graphique est situé "en dessous" du dessin des contrôles.

```
using( Graphics g  =  panel1.CreateGraphics ( ) ) {
  g.FillEllipse ( Brushes.MediumSlateBlue,5,10,40,30 );
}
```



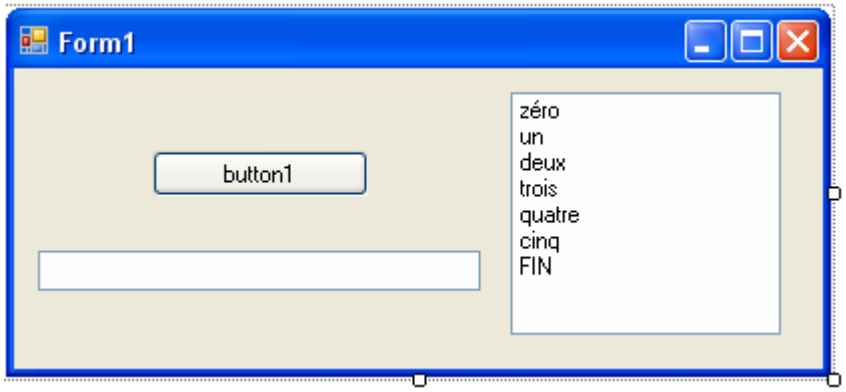
L'instruction dessine l'ellipse bleue à gauche sur le fond du panel1



La première instruction dessine l'ellipse rouge, la seconde écrit le texte "Bonjour" en vert sur le fond du panel2.

Deuxième exemple : traçé des dessins sur des événements spécifiques

Le deuxième exemple montre que l'on peut dessiner aussi sur le fond d'autres contrôles différents des contrôles conteneurs; nous dessinons deux rectangles vert et bleu sur le fond du formulaire et un petit rectangle bleu ciel dans un ListBox, un TextBox et Button déposés sur le formulaire :



Le code de Form1.Designer.cs est le suivant :

```
namespace WindowsApplication3
{
   partial class Form1
   {
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
      protected override void Dispose(bool disposing)
      {
         if (disposing && (components != null))
         {
            components.Dispose();
         }
         base.Dispose(disposing);
      }

      #region Windows Form Designer generated code

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.button1 = new System.Windows.Forms.Button();
         this.listBox1 = new System.Windows.Forms.ListBox();
         this.textBox1 = new System.Windows.Forms.TextBox();
         this.SuspendLayout();
         //
         // button1
         //
         this.button1.Location = new System.Drawing.Point(69, 41);
         this.button1.Name = "button1";
         this.button1.Size = new System.Drawing.Size(108, 23);
         this.button1.TabIndex = 0;
         this.button1.Text = "button1";
         this.button1.UseVisualStyleBackColor = true;
         //
         // listBox1
         //
         this.listBox1.FormattingEnabled = true;
         this.listBox1.Items.AddRange(new object[] {
         "zéro",
         "un",
         "deux",
         "trois",
         "quatre",
         "cinq",
         "FIN"});
         this.listBox1.Location = new System.Drawing.Point(248, 12);
         this.listBox1.Name = "listBox1";
         this.listBox1.Size = new System.Drawing.Size(135, 121);
         this.listBox1.TabIndex = 1;
         //
         // textBox1
```

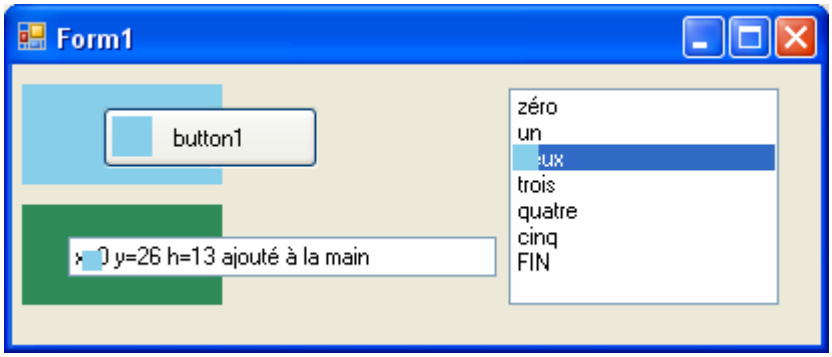
```
            //
            this.textBox1.Location = new System.Drawing.Point(12, 91);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(221, 20);
            this.textBox1.TabIndex = 2;
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(404, 150);
            this.Controls.Add(this.textBox1);
            this.Controls.Add(this.listBox1);
            this.Controls.Add(this.button1);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.ListBox listBox1;
        private System.Windows.Forms.TextBox textBox1;
    }
}
```

Les graphiques sont gérés dans Form1.cs avec le code ci-après :

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

namespace WindowsApplication3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        //-- dessin persistant sur le fond de la fiche par gestionnaire Paint:
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            g.FillRectangle(Brushes.SeaGreen, 5, 70, 100, 50);
            g.FillRectangle(Brushes.SkyBlue, 5, 10, 100, 50);
        }
        //-- dessin persistant sur le fond du bouton par gestionnaire Paint:
        private void button1_Paint(object sender, PaintEventArgs e)
        {
            Graphics x = e.Graphics;
            x.FillRectangle(Brushes.SkyBlue, 5, 5, 20, 20);
        }
```

```
        //-- dessin non persistant sur le fond du ListBox par gestionnaire SelectedIndexChange :
        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            Rectangle Rect = listBox1.GetItemRectangle(listBox1.SelectedIndex);
            int Haut = listBox1.ItemHeight;
            textBox1.Text = "x=" + Rect.X.ToString() + " y=" + Rect.Y.ToString() + " h=" + Haut.ToString();
            using (Graphics k = listBox1.CreateGraphics())
            {
                k.FillRectangle(Brushes.SkyBlue, Rect.X, Rect.Y, Haut, Haut);
            }
        }

        //-- dessin non persistant sur le fond du TextBox par gestionnaire TextChanged :
        private void textBox1_TextChanged(object sender, EventArgs e)
        {
            using (Graphics k = textBox1.CreateGraphics())
            {
                k.FillRectangle(Brushes.SkyBlue, 5, 5, 10, 10);
            }
        }
    }
}
```

Après exécution, sélection de la 3ème ligne de la liste et ajout d'un texte au clavier dans le TextBox :



Exceptions : C# comparé à Java et Delphi

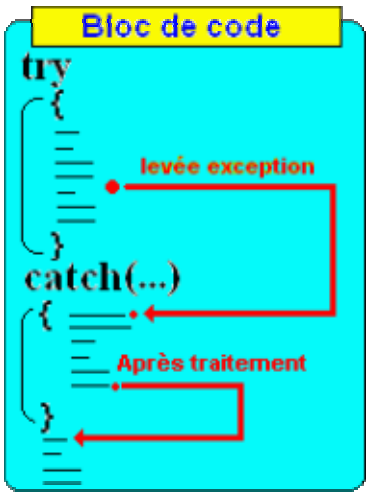
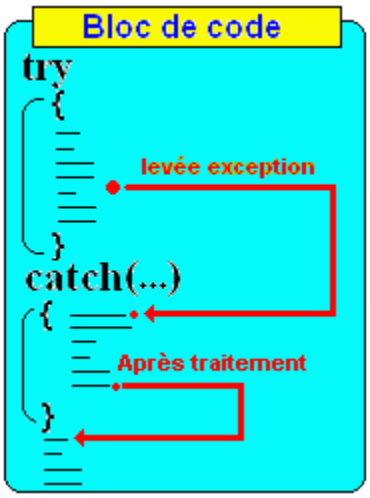
1. similitude et différence

Le langage C# hérite strictement de Java pour la syntaxe et le fonctionnement de base des exceptions et de la simplicité de Delphi dans les types d'exceptions.

Pour une étude complète de la notion d'exception, de leur gestion, de la hiérarchie, de l'ordre d'interception et du redéclenchement d'une exception, nous renvoyons le lecteur au chapitre Traitement d'exceptions du présent ouvrage. Nous figurons ci-dessous un tableau récapitulant les similitudes dans chacun des trois langages :

Delphi	Java	C#

try
- ...
<lignes de code à protéger>
- ...
except
on E : ExxException **do begin**
- ...
<lignes de code réagissant à l'exception>
- ...
end;
- ...
end ;
	try
	{ - ...
	<lignes de code à protéger>
	- ...}
	catch (ExxException E)
	{- ...
	<lignes de code réagissant à l'exception>
	- ...
	}
		try
		{ - ...
		<lignes de code à protéger>
		- ...}
		catch (ExxException E)
		{- ...
		<lignes de code réagissant à l'exception>
		- ...
		}

	fonctionnement identique à C# 	**fonctionnement identique à Java** 

ATTENTION DIFFERENCE : C# - Java

Seul Java possède deux catégories d'exceptions : les exceptions vérifiées et les exceptions non vérifiées. C# comme Delphi possède un mécanisme plus simple qui est équivalent à celui de Java dans le cas des exceptions non vérifiées (implicites) (la propagation de l'exception est **implicitement** prise en charge par le système d'exécution)**.**

2. Créer et lancer ses propres exceptions

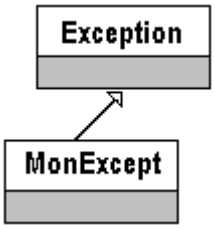
Dans les 3 langages la classes de base des exceptions se nomme : **Exception**

- Il est possible de construire de nouvelles classes d'exceptions personnalisées en héritant d'une des classes du langage, ou à minima de la classe de base **Exception**.

- Il est aussi possible de lancer une exception personnalisée (comme si c'était une exception propre au langage) à n'importe quel endroit dans le code d'une méthode d'une classe, en instanciant un objet d'exception personnalisé et en le préfixant par le mot clef (**raise** pour Delphi et **throw** pour Java et C#).

- Le mécanisme général d'interception des exceptions à travers des gestionnaires d'exceptions **try…except** ou **try…catch** s'applique à tous les types d'exceptions y compris les exceptions personnalisées.

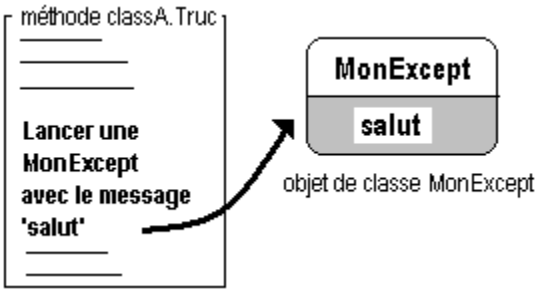
1°) Création d'une nouvelle classe :



Delphi	Java	C#

Type
MonExcept = **class** (Exception)
....
End;
 	
MonExcept hérite par construction de la propriété **public** Message de sa mère, en lecture et écriture.
	class MonExcept **extends** Exception
	{
	public MonExcept (String s)
	{
	super(s);
	.....
	}
	}
		class MonExcept : Exception
		{
		public MonExcept (string s) : **base**(s)
		{
		.....
		}
		}
		MonExcept hérite par construction de la propriété **public** Message de sa mère, en lecture seulement.

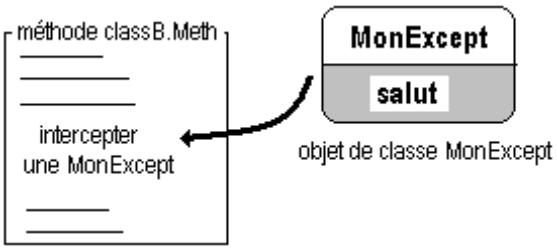
2°) Lancer une MonExcept :



Delphi	Java	C#

Type
MonExcept = **class** (Exception)
.....
End;
 	
classA = **class**
Procedure Truc;
end;
 	
Implementation
 	
Procedure classA.Truc;
begin
…..
raise MonExcept.Create ('salut');
…..
end;
	class MonExcept **extends** Exception
	{
	public MonExcept (String s) {
	super(s);
	.....
	}
	}
	
	class classA
	{
	void Truc ( ) **throws** MonExcept {
	…..
	throw **new** MonExcept ('salut');
	…..
	}
	}
		class MonExcept : Exception
		{
		public MonExcept (string s) : **base**(s)
		{
		.....
		}
		}
		
		class classA
		{
		void Truc ( ) {
		…..
		throw **new** MonExcept ('salut');
		…..
		}
		}

3°) Intercepter une MonExcept :



Delphi	Java	C#

Type
MonExcept = **class** (Exception)
.....
End;
 	
classB = **class**
Procedure Meth;
end;
 	
Implementation
	class MonExcept **extends** Exception
	{
	public MonExcept (String s) {
	super(s);
	.....
	}
	}
	
	class classB
	{
	void Meth ( ) {
		class MonExcept : Exception
		{
		public MonExcept (string s) : **base**(s)
		{
		.....
		}
		}
		
		class classB
		{
		void Meth ( ) {

Procedure classB.Meth;
begin
…..
try
….. // code à protéger
except on MonExcept **do**
begin
….. // code de réaction à l'exception
end;
end;
 	
end;
	…..
	try
	{
	….. // code à protéger
	}
	catch (MonExcept e) {
	….. // code de réaction à l'exception
	}
	}
		…..
		try
		{
		….. // code à protéger
		}
		catch (MonExcept e) {
		….. // code de réaction à l'exception
		}
		}

Données simples et fichiers

Plan général:

1. Les manipulations disque de fichiers et de répertoires

1.1 La manipulation des répertoires
1.2 La manipulation des chemins d'accès disque
1.3 La manipulation des fichiers

2. Flux et fichiers dans NetFramework

2.1 Les flux avec C#
2.2 Les flux et les fichiers de caractères avec C#
2.3 Création d'un fichier de texte avec des données
2.4 Copie des données d'un fichier texte dans un autre

3. Exercice de fichier de salariés avec une IHM

Différences entre flux et fichiers

- Un **fichier** est une ensemble de données structurées possédant un nom (le nom du fichier) stockées généralement sur disque (dans une mémoire de masse en général) dans des enregistrement sous forme d'octets. Les fichiers sont en écriture ou en lecture et ont des organisations diverses comme l'accès séquentiel, l'accès direct, l'accès indexé, en outre les fichiers sont organisés sur le disque dans des répertoires nommés selon des chemins d'accès.
- Un **flux** est un tube de connexion entre deux entrepôts pouvant contenir des données. Ces entités entreposant les données peuvent être des mémoire de masse (disques) ou des zones différentes de la mémoire centrale, ou bien des réseaux différents (par exemple échanges de données entre sockets).
- En C#, il faut utiliser un flux pour accèder aux données d'un fichier stocké sur disque, ce flux connecte les données du fichier sur le disque à une zone précise de la mémoire : le tampon mémoire.

Le Framework.Net dispose d'une famille de classes permettant la manipulation de données externes : **System.IO**

Espace de noms System.IO

- contient des classes qui permettent la lecture et l'écriture dans des fichiers,
- contient des classes qui permettent la lecture et l'écriture dans des flux ,
- contient des classes qui permettent la création, le renommage, le déplacement de fichiers et de répertoires sur disque.

1. Les manipulations disque de fichiers et de répertoires

L'espace de nom **System.IO** contient plusieurs classes de manipulation des fichiers et des répertoires. Certaines ne contiennent que des méthodes **static** (rassemblement de méthodes utiles à la manipulation de fichiers ou de répertoires quelconques), d'autres contiennet des méthodes d'instances et sont utiles lorsque l'on veut travailler sur un objet de fichier précis.

System.IO.Directory **System.IO.DirectoryInfo**
System.IO.File **System.IO.FileInfo**
System.IO.Path

1.1 La manipulation des répertoires

La classe **System.IO.Directory** hérite directement de **System.Object** et ne rajoute que des méthodes **static** pour la création, la suppression, le déplacement et le positionnement de répertoires :

CreateDirectory **Delete** **Exists** **GetAccessControl** **GetCreationTime** **GetCreationTimeUtc** **GetCurrentDirectory** **GetDirectories** **GetDirectoryRoot**	**GetFiles** **GetFileSystemEntries** **GetLastAccessTime** **GetLastAccessTimeUtc** **GetLastWriteTime** **GetLastWriteTimeUtc** **GetLogicalDrives** **GetParent** **Move**	**SetAccessControl** **SetCreationTime** **SetCreationTimeUtc** **SetCurrentDirectory** **SetLastAccessTime** **SetLastAccessTimeUtc** **SetLastWriteTime** **SetLastWriteTimeUtc**

a) *Création d'un répertoire sur disque, s'il n'existe pas au préalable :*

```
string cheminRep = @"D:\MonRepertoire";
if ( !Directory.Exists( cheminRep )) {
        Directory.CreateDirectory( cheminRep );
}
```

Ces 3 lignes de code créent un répertoire MonRepertoire à la racine du disque D.

b) Suppression d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = @"D:\MonRepertoire";
if ( Directory.Exists( cheminRep )) {
        Directory.Delete( cheminRep, true );
}
```

Ces 3 lignes de code suppriment sur le disque D, le répertoire MonRepertoire et tous ses sous-répertoires.

c) Déplacement d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = "D:\\MonRepertoire";
string cheminDest = "C:\\NouveauRepertoire";
if ( Directory.Exists( cheminRep )) {
        Directory.Move( cheminRep, cheminDest );
}
```

Ces 3 lignes de code déplacent le répertoire MonRepertoire du disque D vers le disque C sous le nouveau nom NouveauRepertoire.

d) Accès au répertoire disque de travail en cours de l'application :

```
string cheminRepAppli = Directory.GetCurrentDirectory( );
```

Attention, ce répertoire est le dernier répertoire en cours utilisé par l'application, il n'est pas nécessairement celui du processus qui a lancé l'application, ce dernier peut être obtenu à partir de la classe **Application** par la propriété "**static string** StartupPath" :

```
string cheminLancerAppli = Application.StartupPath ;
```

La classe **System.IO.DirectoryInfo** qui hérite directement de **System.Object** sert aussi à la création, à la suppression, au déplacement et au positionnement de répertoires, par rapport à la classe Directory, certaines méthodes sont transformées en propriétés afin d'en avoir une utilisation plus souple. Conseil de Microsoft : "*si vous souhaitez réutiliser un objet plusieurs fois, l'utilisation de la méthode d'instance de **DirectoryInfo** à la place des méthodes **static** correspondantes de la classe **Directory** peut être préférable*". Le lecteur choisira selon l'application qu'il développe la classe qui lui procure le plus de confort. Afin de bien comparer ces deux classes nous reprenons avec la classe **DirectoryInfo**, les mêmes exemples de base de la classe **Directory**.

a) *Création d'un répertoire sur disque, s'il n'existe pas au préalable :*

```
string cheminRep = @"D:\MonRepertoire";
DirectoryInfo Repertoire = new DirectoryInfo( cheminRep );

if ( !Repertoire.Exists ) {
        Repertoire.Create( );
}
```

Ces lignes de code créent un répertoire MonRepertoire à la racine du disque D.

b) Suppression d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = "D:\\MonRepertoire";
DirectoryInfo Repertoire = new DirectoryInfo( cheminRep );

if ( Repertoire.Exists ) {
        Repertoire.Delete( true );
}
```

Ces lignes de code suppriment sur le disque D, le répertoire MonRepertoire et tous ses sous-répertoires.

c) Déplacement d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = @"D:\MonRepertoire";
string cheminDest = @"C:\NouveauRepertoire";
```

```
DirectoryInfo Repertoire = new DirectoryInfo( cheminRep );

if (  Repertoire.Exists ) {
            Repertoire.MoveTo( cheminDest );
}
```

Ces lignes de code déplacent le répertoire MonRepertoire du disque D vers le disque C sous le nouveau nom NouveauRepertoire.

1.2 La manipulation des chemins d'accès disque

La classe **System.IO.Path** hérite directement de **System.Object** et ne rajoute que des méthodes et des champs **static** pour la manipulation de chaînes **string** contenant des chemins d'accès disque (soit par exemple à partir d'une **string** contenant un chemin comme "c:\rep1\rep2\rep3\appli.exe", extraction du nom de fichier appli.exe, du répertoire c:\rep1\rep2\rep3, de l'extension .exe, etc...).

Ci-dessous nous donnons quelques méthodes **static** pratiques courantes de la classe permettant les manipulations usuelles d'un exemple de chemin d'accès disque sous Windows :

```
string chemin = "C:\\rep1\\rep2\\appli.exe";
ou bien
string chemin = @"C:\rep1\rep2\appli.exe";
```

Appel de la méthode static sur la string chemin	résultat obtenu après l'appel
System.IO.Path.GetDirectoryName (chemin)	C:\rep1\rep2
System.IO.Path.GetExtension (chemin)	.exe
System.IO.Path.GetFileName (chemin)	appli.exe
System.IO.Path.GetFileNameWithoutExtension (chemin)	appli
System.IO.Path.GetFullPath ("C:\\rep1\\rep2\\..\\appli.exe")	C:\rep1\appli.exe
System.IO.Path.GetPathRoot (chemin)	C:\

1.3 La manipulation des fichiers

La classe **System.IO.File** hérite directement de **System.Object** et ne rajoute que des méthodes **static** pour la manipulation, la lecture et l'écriture de fichiers sur disque :

AppendAll	**GetLastAccessTime**	**SetAccessControl**
AppendAllText	**GetLastAccessTimeUtc**	**SetAttributes**
AppendText	**GetLastWriteTime**	**SetCreationTime**
Copy	**GetLastWriteTimeUtc**	**SetCreationTimeUtc**
Create	**Move**	**SetLastAccessTime**
CreateText	**Open**	**SetLastAccessTimeUtc**
Decrypt	**OpenRead**	**SetLastWriteTime**

Delete **Encrypt** **Exists** **GetAccessControl** **GetAttributes** **GetCreationTime** **GetCreationTimeUtc**	**OpenText** **OpenWrite** **ReadAll** **ReadAllBytes** **ReadAllLines** **ReadAllText** **Replace**	**SetLastWriteTimeUtc** **WriteAll** **WriteAllBytes** **WriteAllLines** **WriteAllText**

Exemple de création d'un fichier et de ses répertoires sur le disque en combinant les classes **Directory**, **Path** et **File** :

```
string cheminRep = @"d:\temp\rep1\fichier.txt";
// création d'un répertoire et d'un sous-répertoire s'ils n'existent pas déjà
if ( !Directory.Exists( Path.GetDirectoryName(cheminRep ) ) )
    Directory.CreateDirectory( Path.GetDirectoryName(cheminRep ) );

// effacer le fichier s'il existe déjà
if (File.Exists(cheminRep ))
    File.Delete(cheminRep );

// création du fichier " fichier.txt " sur le disque D : dans " D:\temp\rep1"
FileStream fs = File.Create( cheminRep );
```

Les lignes de code C# précédentes créent sur le disque dur **D:** , les répertoires **temp** et **rep1**, puis le **fichier.txt** selon l'architecture disque ci-après :

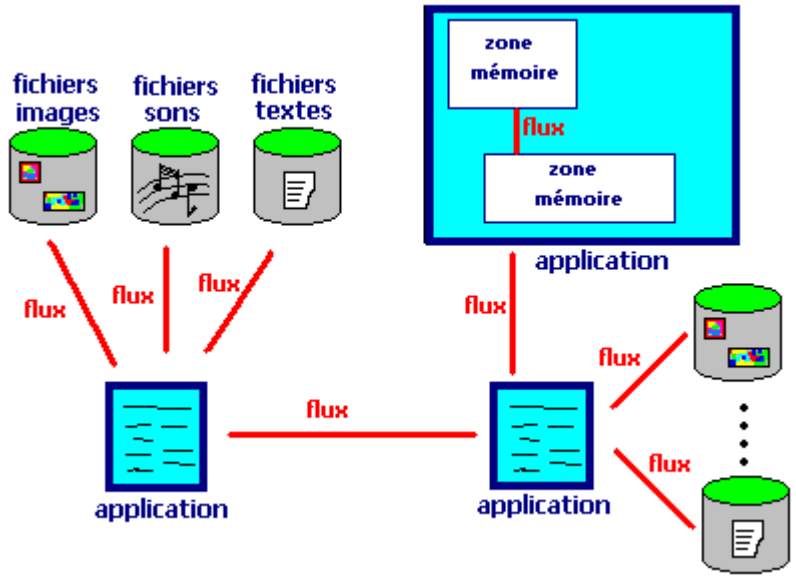
D:

2. Flux et fichiers dans NetFramework

Une application travaille avec ses données internes, mais habituellement il lui est très souvent nécessaire d'aller chercher en **entrée**, on dit **lire**, des nouvelles données (texte, image, son,...) en provenance de diverses sources (périphériques, autres applications, zones mémoires...).

Réciproquement, une application peut après traitement, délivrer en **sortie** des résultats, on dit

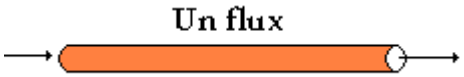
écrire, dans un fichier, vers une autre application ou dans une zone mémoire. **Les flux servent à connecter entre elles ces diverses sources de données**.



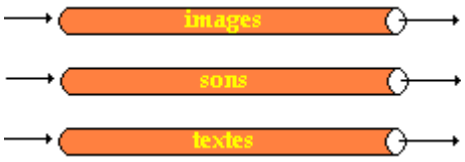
2.1 Les flux avec C#

En C# toutes ces données sont échangées en entrée et en sortie à travers des flux (Stream).

Un flux est une sorte de tuyau de transport séquentiel de données.



Il existe un flux par type de données à transporter :



Un flux est ***unidirectionnel*** : il y a donc des flux d'**entrée** et des flux de **sortie**.

L'image ci-après montre qu'afin de jouer un son **stocké dans un fichier**, l'application C# ouvre en entrée, un flux associé au fichier de sons et lit ce flux séquentiellement afin ensuite de traiter les sons (modifier ou jouer le son) :

La même application peut aussi traiter des images à partir d'un fichier d'images et renvoyer ces images dans le fichier après traitement. C# ouvre un flux en entrée sur le fichier image et un flux en sortie sur le même fichier, l'application lit séquentiellement le flux d'entrée (octet par octet par exemple) et écrit séquentiellement dans le flux de sortie :

D'une manière générale, NetFramework met à notre disposition dans l'espace de noms **System.IO** une classe abstraite de flux de données considérées comme des séquences d'octets, la classe **System.IO.Stream**. Un certain nombre de classes dérivent de la classe **Stream** et fournissent des implémentations spécifiques en particulier les classes :

Classe NetFramework	Utilisation pratique
System.IO.FileStream	consacrée aux flux connectés sur des fichiers.
System.IO.MemoryStream	consacrée aux flux connectés sur des zones mémoires.
System.Net.Sockets.NetworkStream	consacrée aux flux connectés sur des accès réseau.

Nous allons plus particulièrement étudier quelques exemples d'utilisation de flux connectés sur des fichiers et plus précisément des fichiers de textes.

2.2 Les flux et les fichiers de caractères avec C#

Eu égard à l'importance des fichiers comportant des données textuelles (à bases de caractères) le NetFramework dispose de classe de flux chargés de l'ecriture et de la lecture de caractères plus spécialisées que la classe **System.IO.FileStream**.

> Les classes de base **abstraites** de ces flux de caractères se dénomment **System.IO.TextReader** pour la classe permettant la création de flux de lecture séquentielle de caractères et **System.IO.TextWriter** pour la classe permettant la création de flux d'écriture séquentielle de caractères.

Les **classes concrètent** et donc pratiques, qui sont fournies par NetFramework et qui implémentent ces deux classes abstraites sont :

System.IO.StreamReader qui dérive et implémente **System.IO.TextReader**.
System.IO.StreamWriter qui dérive et implémente **System.IO.TextWriter**.

Entrées/Sorties synchrone - asynchrone

synchrone

Les accès des flux aux données (lecture par **Read** ou écriture de base par **Write**) sont par défaut en mode **synchrone** c'est à dire : la méthode qui est en train de lire ou d'écrire dans le flux elle ne peut exécuter aucune autre tâche jusqu'à ce que l'opération de lecture ou d'écriture soit achevée. Le traitement des entrées/sorties est alors "**séquentiel**".

asynchrone

Ceci peut être pénalisant si l'application travaille sur de grandes quantités de données et surtout lorsque plusieurs entrées/sorties doivent avoir lieu "en même temps", dans cette éventualité NetFramework fournit des entrées/sorties multi-threadées (qui peuvent avoir lieu "en même temps") avec les méthodes **asynchrones** de lecture **BeginRead** et **EndRead** et d'écriture **BeginWrite** et **EndWrite**). Dans le cas d'utilisation de méthodes asynchrones, le thread principal peut continuer à effectuer d'autres tâches : le traitement des entrées/sorties est alors "**parallèle**".

Tampon (Buffer)

Les flux du NetFramework sont tous par défauts "bufférisés" contrairement à Java. Dans le NetFramework, un flux (un objet de classe Stream) est une abstraction d'une séquence d'octets, telle qu'un fichier, un périphérique d'entrée/sortie, un canal de communication à processus interne, ou un socket TCP/IP. En C#, un objet flux de classe Stream possède une propriété de longueur **Length** qui indique combien d'octets peuvent être traités par le flux. En outre un objet flux possède aussi une méthode **SetLength**(**long** val) qui définit la longueur du flux : cette mémoire intermédiaire associée au flux est aussi appelée **mémoire tampon** (**Buffer** en Anglais) du flux.

Lorsqu'un flux travaille sur des caractères, on peut faire que l'application lise ou écrive les caractères les uns après les autres en réglant la longueur du flux à 1 caractère (correspondance avec les flux non bufférisés de Java) :

On peut aussi faire que l'application lise ou écrive les caractères par groupes en réglant la longueur du flux à n caractères (correspondance avec les flux bufférisés de Java) :

2.3 Création d'un fichier de texte avec des données

Exemple d'utilisation des 2 classes précédentes.

Supposons que nous ayons l'architecture suivante sur le disque C:

Il est possible de créer un nouveau fichier sur le disque dur et d'ouvrir un flux en écriture sur ce fichier en utilisant le constructeur de la classe **System.IO.StreamWriter** :

```
StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\essai.txt");
```

Voici le résultat obtenu par la ligne de code précédente sur le disque dur :

Le programme C# ci-dessous permet de créer le fichier texte nommé essai.txt et d'écrire des lignes de texte dans ce fichier avec la méthode WriteLine(...) de la classe **StreamWriter**, puis de lire le contenu selon le schéma ci-après avec la méthode ReadLine( ) de la classe **StreamReader** :

Code source C# correspondant :

```
if (!File.Exists( @"c:\temp\rep1\essai.txt" ))    {
    // création d'un fichier texte et ouverture d'un flux en écriture sur ce fichier
    StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\essai.txt");
    Console.WriteLine("Fichier essai.text créé sur le disque dur.");
```

```
        Console.WriteLine("Il n'écrase pas les données déjà présentes");

        // écriture de lignes de texte dans le fichier à travers le flux :
         for ( int i = 1; i<10; i++)
                fluxWrite.WriteLine("texte stocké par programme ligne N : "+i);
          fluxWrite.Close( ); // fermeture du flux impérative pour sauvegarde des données
    }

    Console.WriteLine("Contenu du fichier essai.txt déjà présent :");
    // création et ouverture d'un flux en lecture sur ce fichier
     StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt");

    // lecture des lignes de texte dans le fichier à travers le flux
    string ligne;
    while ( ( ligne = fluxRead.ReadLine()) != null )
        Console.WriteLine(ligne);
    fluxRead.Close();// fermeture du flux
```

2.4 Copie des données d'un fichier texte dans un autre

Nous utilisons l'instruction **using** qui définit un bloc permettant d'instancier un objet local au bloc à la fin duquel un objet est désalloué.

Un bloc **using** instanciant un objet de flux en lecture :
using (**StreamReader** fluxRead = **new** StreamReader(...)) { ....*BLOC*.... }

Un bloc **using** instanciant un objet de flux en écriture :
using (StreamWriter fluxWrite = **new** StreamWriter(...)) { ....*BLOC*.... }

Code source C# créant le fichier essai.txt :

```
    if ( !File.Exists( @"c:\temp\rep1\essai.txt" ))    {

        using (StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\essai.txt"))
        {
            Console.WriteLine("Fichier essai.text créé sur le disque dur.");
            Console.WriteLine("Il n'écrase pas les données déjà présentes");
            // écriture de lignes de texte dans le fichier à travers le flux :
            for ( int i = 1; i<10; i++)
               fluxWrite.WriteLine("texte stocké par programme ligne N : "+i);
         } // fermeture et désallocation de l'objet fluxWrite
    }
```

Code source C# lisant le contenu du fichier essai.txt :

```
   Console.WriteLine("Contenu du fichier 'essai.text' déjà présent :");
   using ( StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt"))
   {
            string ligne;
```

```
            // lecture des lignes de texte dans le fichier à travers le flux :
            while ((ligne = fluxRead.ReadLine()) !=  null )
                Console.WriteLine(ligne);
            Console.WriteLine("\nRecopie en cours ...");
    } // fermeture et désallocation de l'objet fluxRead
```

Code source C# recopiant le contenu du fichier essai.txt, créant le fichier CopyEssai.txt et recopiant le contenu de essai.txt :

```
using ( StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt"))
{
    StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\CopyEssai.txt");
    string ligne;
   // on lit dans essai.txt à travers fluxRead et on écrit dans
   //  CopyEssai.txt à travers fluxWrite :
    while ((ligne = fluxRead.ReadLine()) != null )
        fluxWrite.WriteLine("copie < "+ligne+" >");
    fluxWrite.Close();// fermeture de fluxWrite
}// fermeture et désallocation de l'objet fluxRead
```

Code source C# lisant le contenu du fichier CopyEssai.txt :

```
using ( StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt"))
{
    Console.WriteLine("\nContenu de la copie 'copyEssai.txt' :");
    string ligne;
   // lecture des lignes de texte du nouveau fichier copié fichier à travers le flux :
    while ((ligne = fluxRead.ReadLine()) != null )
        Console.WriteLine(ligne);
}    // fermeture et désallocation de l'objet fluxRead
```

Résultat d'exécution du code précédent :

Contenu du fichier 'essai.text' déjà présent :
texte stocké par programme ligne N : 1
texte stocké par programme ligne N : 2
.....
texte stocké par programme ligne N : 9

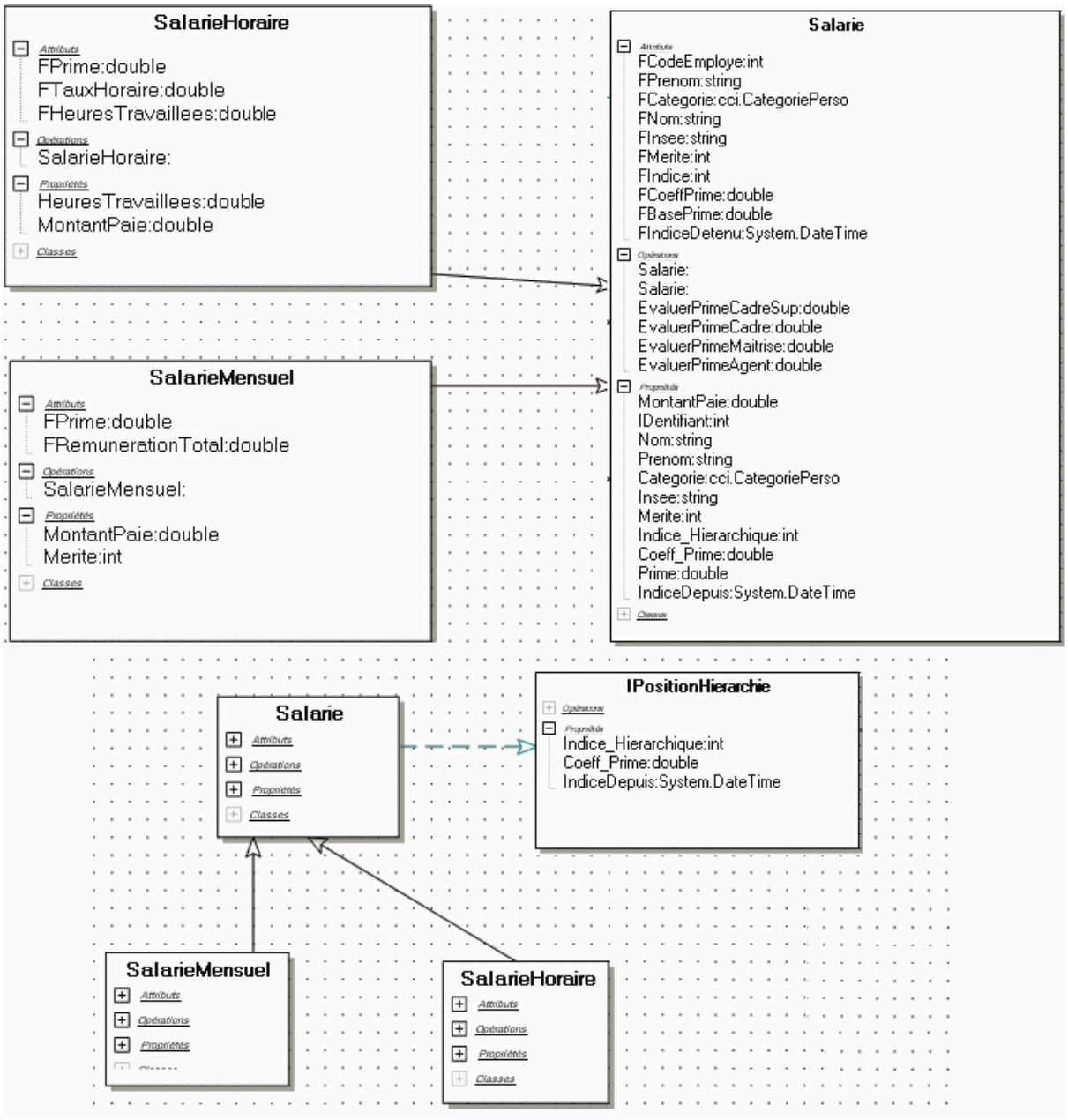
Recopie en cours ...
Contenu de la copie 'copyEssai.txt' :
copie < texte stocké par programme ligne N : 1 >
copie < texte stocké par programme ligne N : 2 >
.....
copie < texte stocké par programme ligne N : 9 >

3. Exercice de fichier de salariés avec une IHM

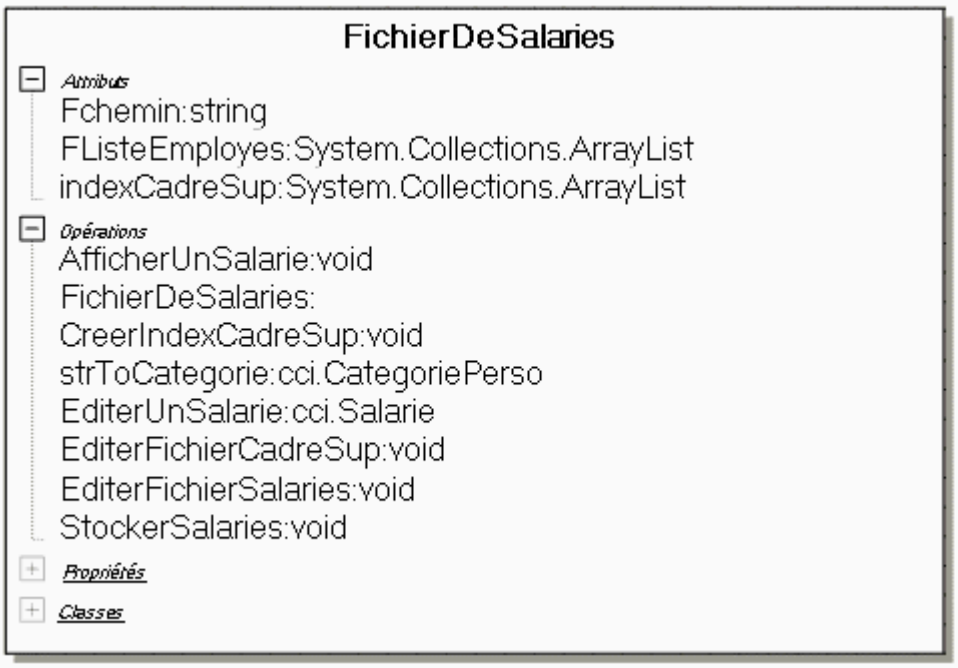
Soit à écrire un programme C# gérant en saisie et en consultation un fichier des salariés d'une petite entreprise.

Ci-dessous les éléments à écrire en mode console d'abord, puis ensuite créez vous-mêmes une interface interactive.

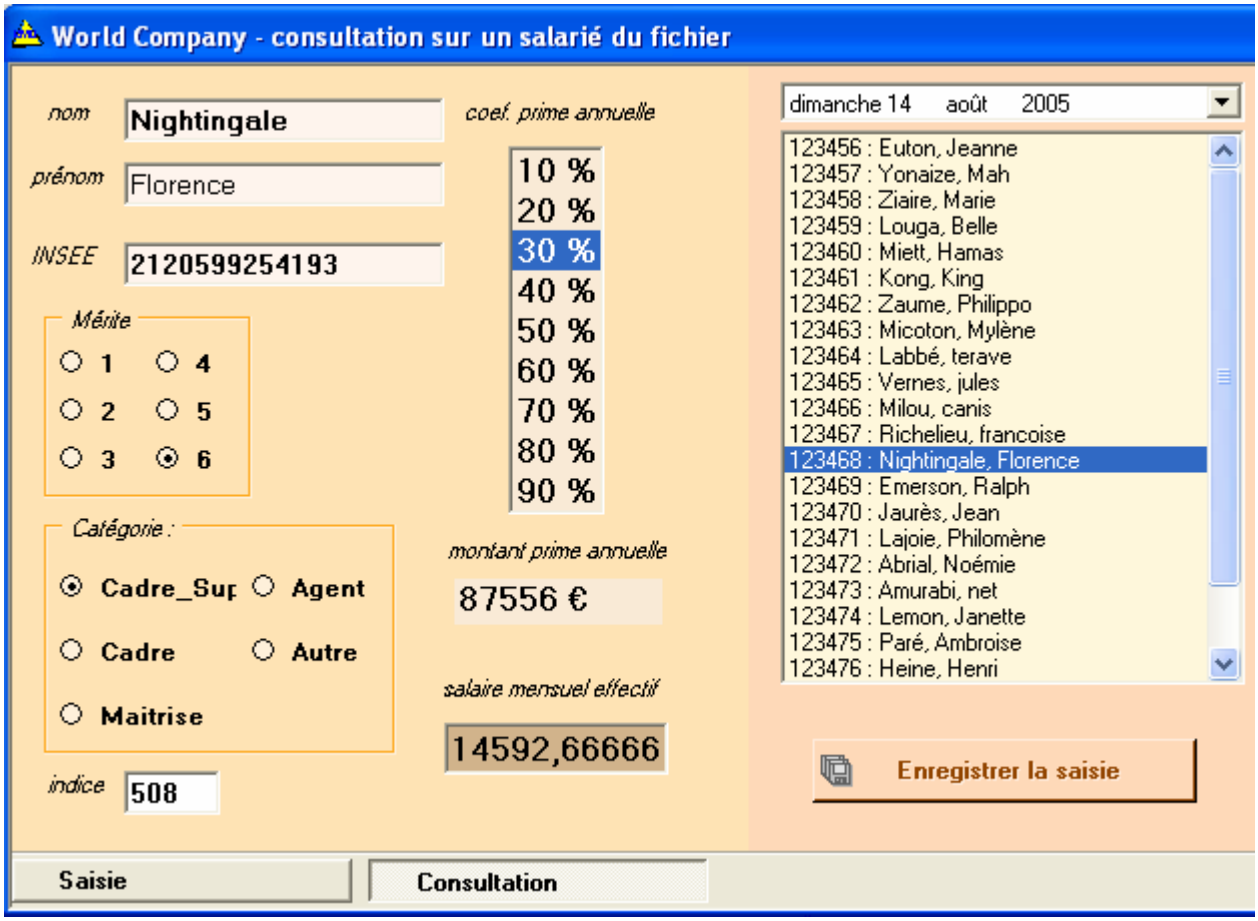
Soit les diagrammes de classe suivants :



Le programme travaille sur un fichier d'objets de classe Salarie :



Un exemple d'IHM possible pour ce programme de gestion de salariés :



Les informations afférentes à un salarié de l'entreprise sont stockées dans un fichier **Fiches** qui est un objet de classe FichierDeSalaries qui se trouve stocké sur le disque dur sous le nom de fichierSalaries.txt. Le programme travaille en mémoire centrale sur une image de ce fichier qui est rangée dans un **ArrayList** nommé ListeSalaries :

```
ArrayList ListeSalaries = new ArrayList ( ) ;

FichierDeSalaries Fiches = new FichierDeSalaries ("fichierSalaries.txt" , ListeSalaries );
```

Squelette et implémentation partielle proposés des classes de base :

```
enum CategoriePerso
{
 Cadre_Sup,Cadre,Maitrise,Agent,Autre
}

/// <summary>
/// Interface définissant les propriétés de position d'un
/// salarié dans la hiérarchie de l'entreprise.
/// </summary>
interface IPositionHierarchie {
 int Indice_Hierarchique
 {
  get ;
  set ;
 }
 double Coeff_Prime
 {
  get ;
  set ;
 }
 DateTime IndiceDepuis
 {
  get ;
  set ;
 }
} // fin interface IPositionHierarchie
/// <summary>
/// Classe de base abstraite pour le personnel. Cette classe n'est
/// pas instanciable.
/// </summary>
abstract class Salarie : IPositionHierarchie {
 /// attributs identifiant le salarié :
 private int FCodeEmploye ;
 private string FPrenom ;
 private CategoriePerso FCategorie ;
 private string FNom ;
 private string FInsee ;
 protected int FMerite ;
```

```
  private int FIndice ;
  private double FCoeffPrime ;
  private double FBasePrime ;
  private DateTime FIndiceDetenu ;

  ///le constructeur de la classe employé au mérite :
  public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie,
  string Insee, int Merite, int Indice, double CoeffPrime )
 {
   FCodeEmploye = IDentifiant ;
   FNom = Nom ;
   FPrenom = Prenom ;
   FCategorie = Categorie ;
   FInsee = Insee ;
   FMerite = Merite ;
   FIndice = Indice ;
   FCoeffPrime = CoeffPrime ;
   FIndiceDetenu = DateTime.Now ;
  switch ( FCategorie ) {
   case CategoriePerso.Cadre_Sup :    FBasePrime = 2000 ;   break;
   case CategoriePerso.Cadre :    FBasePrime = 1000 ;   break;
   case CategoriePerso.Maitrise :    FBasePrime = 500 ;   break;
   case CategoriePerso.Agent :    FBasePrime = 200 ;   break;
  }
 }
  ///le constructeur de la classe employé sans mérite :
 public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie, string
Insee ): this( IDentifiant, Nom, Prenom, Categorie, Insee,0,0,0 ) {


}

 protected double EvaluerPrimeCadreSup ( int coeffMerite ) {
  return ( 100 + coeffMerite * 8 ) * FCoeffPrime * FBasePrime + FIndice * 7 ;
 }
 protected double EvaluerPrimeCadre ( int coeffMerite ) {
  return ( 100 + coeffMerite * 6 ) * FCoeffPrime * FBasePrime + FIndice * 5 ;
 }
 protected double EvaluerPrimeMaitrise ( int coeffMerite ) {
  return ( 100 + coeffMerite * 4 ) * FCoeffPrime * FBasePrime + FIndice * 3 ;
 }
 protected double EvaluerPrimeAgent ( int coeffMerite ) {
  return ( 100 + coeffMerite * 2 ) * FCoeffPrime * FBasePrime + FIndice * 2 ;
 }
  /// propriété abstraite donnant le montant du salaire
 /// (virtual automatiquement)
  abstract public double MontantPaie  {   get ; }
  /// propriété identifiant le salarié dans l'entreprise :
  public int IDentifiant  {
   get   {   return FCodeEmploye ;   }
 }
  /// propriété nom du salarié :
```

```
 public string Nom {
  get { return FNom ; }
  set { FNom = value ; }
 }
 /// propriété nom du salarié :
 public string Prenom {
  get { return FPrenom ; }
  set { FPrenom = value ; }
 }
 /// propriété catégorie de personnel du salarié :
 public CategoriePerso Categorie {
  get { return FCategorie ; }
  set { FCategorie = value ; }
 }
 /// propriété n°; de sécurité sociale du salarié :
 public string Insee {
  get { return FInsee ; }
  set { FInsee = value ; }
 }
 /// propriété de point de mérite du salarié :
 public virtual int Merite {
  get { return FMerite ; }
  set { FMerite = value ; }
 }
 /// propriété classement indiciaire dans la hiérarchie :
 public int Indice_Hierarchique {
  get { return FIndice ; }
  set {
   FIndice = value ;
   //--Maj de la date de détention du nouvel indice :
   IndiceDepuis = DateTime.Now ;
  }
 }
 /// propriété coefficient de la prime en %:
 public double Coeff_Prime {
  get { return FCoeffPrime ; }
  set { FCoeffPrime = value ; }
 }
 /// propriété valeur de la prime :
 public double Prime {
  get {
    switch ( FCategorie )
    {
      case CategoriePerso.Cadre_Sup : return EvaluerPrimeCadreSup ( FMerite );
      case CategoriePerso.Cadre : return EvaluerPrimeCadre ( FMerite );
      case CategoriePerso.Maitrise : return EvaluerPrimeMaitrise ( FMerite );
      case CategoriePerso.Agent : return EvaluerPrimeAgent ( FMerite );
      default : return EvaluerPrimeAgent ( 0 );
    }
  }
 }
```

```
  /// date à laquelle l'indice actuel a été obtenu :
  public DateTime IndiceDepuis  {
   get  {  return FIndiceDetenu ;  }
   set  {   FIndiceDetenu = value ;  }
  }
 } // fin classe Salarie

  /// <summary>
 /// Classe du personnel mensualisé. Implémente la propriété abstraite
 /// MontantPaie déclarée dans la classe de base (mère).
 /// </summary>
  class SalarieMensuel : Salarie
  {
   /// attributs du salaire annuel :
   private double FPrime ;
   private double FRemunerationTotal ;

   ///le constructeur de la classe (salarié au mérite) :
  public SalarieMensuel ( int IDentifiant, string Nom, string Prenom, CategoriePerso
Categorie, string Insee, int Merite, int Indice, double CoeffPrime, double
RemunerationTotal ) :base ( IDentifiant, Nom, Prenom, Categorie, Insee, Merite, Indice,
CoeffPrime )
  {
    FPrime = this .Prime ;
    FRemunerationTotal = RemunerationTotal ;
  }
   /// implémentation de la propriété donnant le montant du salaire :
   public override double MontantPaie  {
    get  {  return ( FRemunerationTotal + this .Prime ) / 12 ;  }
  }


 /// propriété de point de mérite du salarié :
   public override int Merite  {
    get  {  return FMerite ;  }
    set  {
            FMerite = value ;
            FPrime = this .Prime ;
         }
   }
 } // fin classe SalarieMensuel

  /// <summary>
 /// Classe du personnel horaire. Implemente la propriété abstraite
 /// MontantPaie déclarée dans la classe de base (mère).
 /// </summary>
  class SalarieHoraire : Salarie
  {
   /// attributs permettant le calcul du salaire :
   private double FPrime ;
   private double FTauxHoraire ;
```

```
  private double FHeuresTravaillees ;

  ///le constructeur de la classe (salarié non au mérite):
public SalarieHoraire ( int IDentifiant, string Nom, string Prenom, string Insee, double
TauxHoraire ): base ( IDentifiant, Nom, Prenom, CategoriePerso.Autre, Insee )
 {
   FTauxHoraire = TauxHoraire ;
   FHeuresTravaillees = 0 ;
   FPrime = 0 ;
 }
  /// nombre d'heures effectuées :
  public double HeuresTravaillees {
   get { return FHeuresTravaillees ; }
   set { FHeuresTravaillees = value ; }
 }
  /// implémentation de la propriété donnant le montant du salaire :
  public override double MontantPaie {
   get { return FHeuresTravaillees * FTauxHoraire + FPrime ; }
 }
} // fin classe SalarieHoraire

class FichierDeSalaries
{
  private string Fchemin ;
  private ArrayList FListeEmployes ; // liste des nouveaux employés à entrer dans le fichier
  private ArrayList indexCadreSup ; // Table d'index des cadres supérieurs du fichier

  // méthode static affichant un objet Salarie à la console :
  public static void AfficherUnSalarie ( Salarie Employe ) {
    // pour l'instant un salarié mensualisé seulement
  }
  // constructeur de la classeFichierDeSalaries
  public FichierDeSalaries ( string chemin, ArrayList Liste ) {

  }
  // méthode de création de la table d'index des cadre_sup :
  public void CreerIndexCadreSup ( ) {

  }
  // méthode convertissant le champ string catégorie en la constante enum associée
  private CategoriePerso strToCategorie ( string s ) {

  }
  // méthode renvoyant un objet SalarieMensuel de rang fixé dans le fichier
  private Salarie EditerUnSalarie ( int rang ) {
   SalarieMensuel perso ;
    ...........
   perso = new SalarieMensuel ( IDentifiant, Nom, Prenom, Categorie, Insee,
            Merite, Indice, CoeffPrime, RemunerationTotal );
    ...........
   return perso ;
```

```
 }
  // méthode affichant sur la console à partir de la table d'index :
  public void  EditerFichierCadreSup ( )
 {
   ...........
  foreach( int  ind  in  indexCadreSup )
  {
    AfficherUnSalarie (  EditerUnSalarie ( ind ) );
  }
   ...........
 }
  // méthode affichant sur la console le fichier de tous les salariés :
  public void  EditerFichierSalaries ( )  {

  }
  // méthode créant et stockant des salariés dans le fichier :
  public void  StockerSalaries ( ArrayList ListeEmploy )
 {
   ...........
   // si le fichier n'existe pas => création du fichier sur disque :
   StreamWriter fichierSortie  =  File.CreateText ( Fchemin );
    fichierSortie.WriteLine ("Fichier des personnels");
    fichierSortie.Close ();
  ...........
   // ajout dans le fichier de toute  la liste  :
   ...........
   foreach( Salarie s  in  ListeEmploy )
   {

   }
  ...........
 }
} // fin classe FichierDeSalaries
```

Implémenter les classes avec le programme de test suivant :

```
 class  ClassUsesSalarie
 {
  /// <summary>
 /// Le point d'entrée principal de l'application.
 /// </summary>
  static void  InfoSalarie ( SalarieMensuel empl )
 {
   FichierDeSalaries.AfficherUnSalarie ( empl );
   double  coefPrimeLoc = empl.Coeff_Prime ;
   int  coefMeriteLoc = empl.Merite ;
   //--impact variation du coef de prime
   for( double  i = 0.5 ;  i < 1 ;  i  += 0.1 )
  {
    empl.Coeff_Prime =  i ;
```

```
  Console .WriteLine ("   coeff prime : " + empl.Coeff_Prime );
  Console .WriteLine ("   montant prime annuelle : " + empl.Prime );
  Console .WriteLine ("   montant paie mensuelle: " + empl.MontantPaie );
 }
 Console .WriteLine ("   -----------------------");
 empl.Coeff_Prime = coefPrimeLoc ;
 //--impact variation du coef de mérite
 for( int  i = 0 ;  i < 10 ;  i  ++ )
 {
  empl.Merite =  i ;
  Console .WriteLine ("   coeff mérite : " + empl.Merite );
  Console .WriteLine ("   montant prime annuelle : " + empl.Prime );
  Console .WriteLine ("   montant paie mensuelle: " + empl.MontantPaie );
 }
 empl.Merite = coefMeriteLoc ;
 Console .WriteLine ("=========================================");

}
 [STAThread]
 static void  Main ( string [] args )
{
  SalarieMensuel Employe1  =  new  SalarieMensuel ( 123456, "Euton" , "Jeanne" ,
      CategoriePerso.Cadre_Sup, "2780258123456" ,6,700,0.5,50000 );
  SalarieMensuel Employe2  =  new  SalarieMensuel ( 123457, "Yonaize" , "Mah" ,
      CategoriePerso.Cadre, "1821113896452" ,5,520,0.42,30000 );
  SalarieMensuel Employe3  =  new  SalarieMensuel ( 123458, "Ziaire" , "Marie" ,
      CategoriePerso.Maitrise, "2801037853781" ,2,678,0.6,20000 );
  SalarieMensuel Employe4  =  new  SalarieMensuel ( 123459, "Louga" , "Belle" ,
      CategoriePerso.Agent, "2790469483167" ,4,805,0.25,20000 );

  ArrayList ListeSalaries  =  new  ArrayList ();
  ListeSalaries.Add (  Employe1  );
  ListeSalaries.Add (  Employe2  );
  ListeSalaries.Add (  Employe3  );
  ListeSalaries.Add (  Employe4  );
 foreach( SalarieMensuel s  in  ListeSalaries )
   InfoSalarie ( s );
  Console .WriteLine (">>>  Promotion indice de " + Employe1.Nom + " dans 2 secondes.");
  Thread.Sleep ( 2000 );
  Employe1.Indice_Hierarchique = 710 ;
  InfoSalarie (  Employe1  );
  //-----------------------------------------//
  FichierDeSalaries Fiches  =  new  FichierDeSalaries ("fichierSalaries.txt" ,ListeSalaries );
  Console .WriteLine (">>>  Attente 3 s pour création de nouveaux salariés");
  Thread.Sleep ( 3000 );
  Employe1  =  new  SalarieMensuel ( 123460, "Miett" , "Hamas" ,
      CategoriePerso.Cadre_Sup, "1750258123456" ,4,500,0.7,42000 );
  Employe2  =  new  SalarieMensuel ( 123461, "Kong" , "King" ,
      CategoriePerso.Cadre, "1640517896452" ,4,305,0.62,28000 );
  Employe3  =  new  SalarieMensuel ( 123462, "Zaume" , "Philippo" ,
      CategoriePerso.Maitrise, "1580237853781" ,2,245,0.8,15000 );
```

```
    Employe4 = new SalarieMensuel ( 123463, "Micoton" , "Mylène" ,
        CategoriePerso.Agent, "2850263483167" ,4,105,0.14,12000 );
    ListeSalaries = new ArrayList ();
    ListeSalaries.Add ( Employe1 );
    ListeSalaries.Add ( Employe2 );
    ListeSalaries.Add ( Employe3 );
    ListeSalaries.Add ( Employe4 );
    Fiches.StockerSalaries ( ListeSalaries );
    Fiches.EditerFichierSalaries ();
    Fiches.CreerIndexCadreSup ();
    Fiches.EditerFichierCadreSup ();
    System .Console.ReadLine ();
  }
 }
}
```

Exemple de résultats obtenus avec le programme de test précédent :

fichierSalaries.txt :

```
Fichier des personnels
123456
Euton
Jeanne
*Cadre_Sup
2780258123456
6
710
15/02/2004 19:52:38
0,5
152970
16914,1666666667
123457
Yonaize
Mah
*Cadre
1821113896452
5
520
15/02/2004 19:52:36
0,42
57200
7266,66666666667
123458
Ziaire
Marie
*Maitrise
2801037853781
2
678
15/02/2004 19:52:36
```

0,6
34434
4536,16666666667
123459
Louga
Belle
*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,83333333333
123460
Miett
Hamas
*Cadre_Sup
1750258123456
4
500
15/02/2004 19:52:41
0,7
188300
19191,6666666667
123461
Kong
King
*Cadre
1640517896452
4
305
15/02/2004 19:52:41
0,62
78405
8867,08333333333
123462
Zaume
Philippo
*Maitrise
1580237853781
2
245
15/02/2004 19:52:41
0,8
43935
4911,25
123463
Micoton
Mylène
*Agent

2850263483167
4
105
15/02/2004 19:52:41
0,14
3234
1269,5

Résultats console :

```
Employé n&deg;123456: Euton / Jeanne
 n&deg; SS : 2780258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 700 , détenu depuis : 15/02/2004 19:52:36
 coeff mérite : 6
 coeff prime : 0,5
 montant prime annuelle : 152900
 montant paie mensuelle: 16908,3333333333
   coeff prime : 0,5
   montant prime annuelle : 152900
   montant paie mensuelle: 16908,3333333333
   coeff prime : 0,6
   montant prime annuelle : 182500
   montant paie mensuelle: 19375
   coeff prime : 0,7
   montant prime annuelle : 212100
   montant paie mensuelle: 21841,6666666667
   coeff prime : 0,8
   montant prime annuelle : 241700
   montant paie mensuelle: 24308,3333333333
   coeff prime : 0,9
   montant prime annuelle : 271300
   montant paie mensuelle: 26775
   coeff prime : 1
   montant prime annuelle : 300900
   montant paie mensuelle: 29241,6666666667
   -----------------------
   coeff mérite : 0
   montant prime annuelle : 104900
   montant paie mensuelle: 12908,3333333333
   coeff mérite : 1
   montant prime annuelle : 112900
   montant paie mensuelle: 13575
   coeff mérite : 2
   montant prime annuelle : 120900
   montant paie mensuelle: 14241,6666666667
   coeff mérite : 3
   montant prime annuelle : 128900
   montant paie mensuelle: 14908,3333333333
   coeff mérite : 4
   montant prime annuelle : 136900
```

```
  montant paie mensuelle: 15575
  coeff mérite : 5
  montant prime annuelle : 144900
  montant paie mensuelle: 16241,6666666667
  coeff mérite : 6
  montant prime annuelle : 152900
  montant paie mensuelle: 16908,3333333333
  coeff mérite : 7
  montant prime annuelle : 160900
  montant paie mensuelle: 17575
  coeff mérite : 8
  montant prime annuelle : 168900
  montant paie mensuelle: 18241,6666666667
  coeff mérite : 9
  montant prime annuelle : 176900
  montant paie mensuelle: 18908,3333333333
========================================
Employé n&deg;123457: Yonaize / Mah
 n&deg; SS : 1821113896452
 catégorie : Cadre
 indice hiérarchique : 520 , détenu depuis : 15/02/2004 19:52:36
 coeff mérite : 5
 coeff prime : 0,42
 montant prime annuelle : 57200
 montant paie mensuelle: 7266,66666666667
  coeff prime : 0,5
  montant prime annuelle : 67600
  montant paie mensuelle: 8133,33333333333
  coeff prime : 0,6
  montant prime annuelle : 80600
  montant paie mensuelle: 9216,66666666667
  coeff prime : 0,7
  montant prime annuelle : 93600
  montant paie mensuelle: 10300
  coeff prime : 0,8
  montant prime annuelle : 106600
  montant paie mensuelle: 11383,3333333333
  coeff prime : 0,9
  montant prime annuelle : 119600
  montant paie mensuelle: 12466,6666666667
  coeff prime : 1
  montant prime annuelle : 132600
  montant paie mensuelle: 13550
  -----------------------
  coeff mérite : 0
  montant prime annuelle : 44600
  montant paie mensuelle: 6216,66666666667
  coeff mérite : 1
  montant prime annuelle : 47120
  montant paie mensuelle: 6426,66666666667
  coeff mérite : 2
```

```
  montant prime annuelle : 49640
  montant paie mensuelle: 6636,66666666667
  coeff mérite : 3
  montant prime annuelle : 52160
  montant paie mensuelle: 6846,66666666667
  coeff mérite : 4
  montant prime annuelle : 54680
  montant paie mensuelle: 7056,66666666667
  coeff mérite : 5
  montant prime annuelle : 57200
  montant paie mensuelle: 7266,66666666667
  coeff mérite : 6
  montant prime annuelle : 59720
  montant paie mensuelle: 7476,66666666667
  coeff mérite : 7
  montant prime annuelle : 62240
  montant paie mensuelle: 7686,66666666667
  coeff mérite : 8
  montant prime annuelle : 64760
  montant paie mensuelle: 7896,66666666667
  coeff mérite : 9
  montant prime annuelle : 67280
  montant paie mensuelle: 8106,66666666667
=======================================
Employé n&deg;123458: Ziaire / Marie
 n&deg; SS : 2801037853781
 catégorie : Maitrise
 indice hiérarchique : 678 , détenu depuis : 15/02/2004 19:52:36
 coeff mérite : 2
 coeff prime : 0,6
 montant prime annuelle : 34434
 montant paie mensuelle: 4536,16666666667
  coeff prime : 0,5
  montant prime annuelle : 29034
  montant paie mensuelle: 4086,16666666667
  coeff prime : 0,6
  montant prime annuelle : 34434
  montant paie mensuelle: 4536,16666666667
  coeff prime : 0,7
  montant prime annuelle : 39834
  montant paie mensuelle: 4986,16666666667
  coeff prime : 0,8
  montant prime annuelle : 45234
  montant paie mensuelle: 5436,16666666667
  coeff prime : 0,9
  montant prime annuelle : 50634
  montant paie mensuelle: 5886,16666666667
  coeff prime : 1
  montant prime annuelle : 56034
  montant paie mensuelle: 6336,16666666667
  -----------------------
```

```
  coeff mérite : 0
  montant prime annuelle : 32034
  montant paie mensuelle: 4336,16666666667
  coeff mérite : 1
  montant prime annuelle : 33234
  montant paie mensuelle: 4436,16666666667
  coeff mérite : 2
  montant prime annuelle : 34434
  montant paie mensuelle: 4536,16666666667
  coeff mérite : 3
  montant prime annuelle : 35634
  montant paie mensuelle: 4636,16666666667
  coeff mérite : 4
  montant prime annuelle : 36834
  montant paie mensuelle: 4736,16666666667
  coeff mérite : 5
  montant prime annuelle : 38034
  montant paie mensuelle: 4836,16666666667
  coeff mérite : 6
  montant prime annuelle : 39234
  montant paie mensuelle: 4936,16666666667
  coeff mérite : 7
  montant prime annuelle : 40434
  montant paie mensuelle: 5036,16666666667
  coeff mérite : 8
  montant prime annuelle : 41634
  montant paie mensuelle: 5136,16666666667
  coeff mérite : 9
  montant prime annuelle : 42834
  montant paie mensuelle: 5236,16666666667
=====================================
Employé n&deg;123459: Louga / Belle
 n&deg; SS : 2790469483167
 catégorie : Agent
 indice hiérarchique : 805 , détenu depuis : 15/02/2004 19:52:36
 coeff mérite : 4
 coeff prime : 0,25
 montant prime annuelle : 7010
 montant paie mensuelle: 2250,83333333333
  coeff prime : 0,5
  montant prime annuelle : 12410
  montant paie mensuelle: 2700,83333333333
  coeff prime : 0,6
  montant prime annuelle : 14570
  montant paie mensuelle: 2880,83333333333
  coeff prime : 0,7
  montant prime annuelle : 16730
  montant paie mensuelle: 3060,83333333333
  coeff prime : 0,8
  montant prime annuelle : 18890
  montant paie mensuelle: 3240,83333333333
```

```
  coeff prime : 0,9
  montant prime annuelle : 21050
  montant paie mensuelle: 3420,83333333333
  coeff prime : 1
  montant prime annuelle : 23210
  montant paie mensuelle: 3600,83333333333
  -----------------------
  coeff mérite : 0
  montant prime annuelle : 6610
  montant paie mensuelle: 2217,5
  coeff mérite : 1
  montant prime annuelle : 6710
  montant paie mensuelle: 2225,83333333333
  coeff mérite : 2
  montant prime annuelle : 6810
  montant paie mensuelle: 2234,16666666667
  coeff mérite : 3
  montant prime annuelle : 6910
  montant paie mensuelle: 2242,5
  coeff mérite : 4
  montant prime annuelle : 7010
  montant paie mensuelle: 2250,83333333333
  coeff mérite : 5
  montant prime annuelle : 7110
  montant paie mensuelle: 2259,16666666667
  coeff mérite : 6
  montant prime annuelle : 7210
  montant paie mensuelle: 2267,5
  coeff mérite : 7
  montant prime annuelle : 7310
  montant paie mensuelle: 2275,83333333333
  coeff mérite : 8
  montant prime annuelle : 7410
  montant paie mensuelle: 2284,16666666667
  coeff mérite : 9
  montant prime annuelle : 7510
  montant paie mensuelle: 2292,5
========================================
>>>  Promotion indice de Euton dans 2 secondes.
Employé n&deg;123456: Euton / Jeanne
 n&deg; SS : 2780258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 710 , détenu depuis : 15/02/2004 19:52:38
 coeff mérite : 6
 coeff prime : 0,5
 montant prime annuelle : 152970
 montant paie mensuelle: 16914,1666666667
  coeff prime : 0,5
  montant prime annuelle : 152970
  montant paie mensuelle: 16914,1666666667
  coeff prime : 0,6
```

```
  montant prime annuelle : 182570
  montant paie mensuelle: 19380,8333333333
  coeff prime : 0,7
  montant prime annuelle : 212170
  montant paie mensuelle: 21847,5
  coeff prime : 0,8
  montant prime annuelle : 241770
  montant paie mensuelle: 24314,1666666667
  coeff prime : 0,9
  montant prime annuelle : 271370
  montant paie mensuelle: 26780,8333333333
  coeff prime : 1
  montant prime annuelle : 300970
  montant paie mensuelle: 29247,5
  -----------------------
  coeff mérite : 0
  montant prime annuelle : 104970
  montant paie mensuelle: 12914,1666666667
  coeff mérite : 1
  montant prime annuelle : 112970
  montant paie mensuelle: 13580,8333333333
  coeff mérite : 2
  montant prime annuelle : 120970
  montant paie mensuelle: 14247,5
  coeff mérite : 3
  montant prime annuelle : 128970
  montant paie mensuelle: 14914,1666666667
  coeff mérite : 4
  montant prime annuelle : 136970
  montant paie mensuelle: 15580,8333333333
  coeff mérite : 5
  montant prime annuelle : 144970
  montant paie mensuelle: 16247,5
  coeff mérite : 6
  montant prime annuelle : 152970
  montant paie mensuelle: 16914,1666666667
  coeff mérite : 7
  montant prime annuelle : 160970
  montant paie mensuelle: 17580,8333333333
  coeff mérite : 8
  montant prime annuelle : 168970
  montant paie mensuelle: 18247,5
  coeff mérite : 9
  montant prime annuelle : 176970
  montant paie mensuelle: 18914,1666666667
========================================
>>>  Attente 3 s pour création de nouveaux salariés
Fichier des personnels
123456
Euton
Jeanne
```

*Cadre_Sup
2780258123456
6
710
15/02/2004 19:52:38
0,5
152970
16914,1666666667
123457
Yonaize
Mah
*Cadre
1821113896452
5
520
15/02/2004 19:52:36
0,42
57200
7266,66666666667
123458
Ziaire
Marie
*Maitrise
2801037853781
2
678
15/02/2004 19:52:36
0,6
34434
4536,16666666667
123459
Louga
Belle
*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,83333333333
123460
Miett
Hamas
*Cadre_Sup
1750258123456
4
500
15/02/2004 19:52:41
0,7
188300

19191,6666666667
123461
Kong
King
*Cadre
1640517896452
4
305
15/02/2004 19:52:41
0,62
78405
8867,08333333333
123462
Zaume
Philippo
*Maitrise
1580237853781
2
245
15/02/2004 19:52:41
0,8
43935
4911,25
123463
Micoton
Mylène
*Agent
2850263483167
4
105
15/02/2004 19:52:41
0,14
3234
1269,5
++> *Cadre_Sup : 5
++> *Cadre_Sup : 49
Employé n°123456: Euton / Jeanne
 n° SS : 2780258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 710 , détenu depuis : 15/02/2004 19:52:38
 coeff mérite : 6
 coeff prime : 0
 montant prime annuelle : 4970
 montant paie mensuelle: 414,208333333333
Employé n°123460: Miett / Hamas
 n° SS : 1750258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 500 , détenu depuis : 15/02/2004 19:52:41
 coeff mérite : 4
 coeff prime : 0
 montant prime annuelle : 3500

montant paie mensuelle: 291,725

Précisons que pour pouvoir utiliser l'instruction de simulation d'attente de 2 secondes entre deux promotions, soit Thread.Sleep **(2000);** il est nécessaire de déclarer dans les clauses using :

using System.Threading;

Les éléments principaux depuis laVersion 2.0

- Les Generics
- Les classes partielles
- Les méthodes anonymes
- TAD de listes, piles, files génériques
- Arbres binaires et classes génériques
- Principes des bases de données
- Ado .net et données relationnelles
- Ado .net et SQL serveur 2005

Les Generics

C# .net 2.0

Plan général:

1. Les generics

1.1 Une liste générale sans les generics
1.2 Une liste générale avec les generics
1.3 Une méthode avec un paramètre générique
1.4 Type générique contraint
1.5 Surcharge générique d'une classe

1. Les generics ou types paramétrés

Les generics introduits depuis la version 2.0 du C# permettent de construire des classes des structs, des interfaces, des delegates ou des méthodes qui sont paramétrés par un type qu'ils sont capables de stocker et de manipuler.

Le compilateur C# reconnaît un type paramétré (generic) lorsqu'il trouve un ou plusieurs identificateurs de type entre crochets <…> :

Exemples de syntaxe de classes génériques

```
//les types paramétrés peuvent être appliqués aux classes aux interfaces
interface IGeneric1<T>
{
}
class ClassGeneric1< UnType, Autre >
{
}
class ClassInt1 : ClassGeneric1< int, int >
{
}
class ClassInt2 <T> : ClassGeneric1< int, T >
{
}
class ClassInt3 <T, U> : ClassGeneric1< int, U >
{
}
```

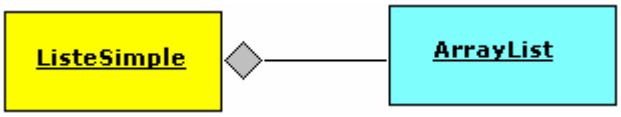
Exemples de syntaxe de méthodes génériques

```
//les types paramétrés peuvent être appliqués aux méthodes
class clA
{
   public void methode1<T>()
   {
   }
   public T[] methode2<T>()
   {
      return new T[10];
   }
}
```

1.1 Une liste générale sans les generics

On pourrait penser que le type object qui est la classe mère des types références et des types valeurs devrait suffire à passer en paramètre n'importe quel type d'objet. Ceci est en effet possible mais avec des risques de mauvais transtypage dont les tests sont à la charge du développeur.

Considérons une classe ListeSimple contenant un objet ArrayList :



Nous souhaitons stocker dans cette classe des entiers de type **int** et afficher un élément de la liste de rang fixé. Nous écrirons un programme dans lequel nous rangeons deux entiers dans un objet listeSimpleInt de type ListeSimple, puis nous les affichons :

```
public class ListeSimple
{
   public ArrayList liste = new ArrayList();

   public void ajouter(object elt)
   {
        this.liste.Add(elt);
   }
   public int afficherInt(int rang)
   {
        return (int)this.liste[rang];
   }
}

class Program
{
   static void Main(string[ ] args)
   {
      ListeSimple listeSimpleInt = new ListeSimple();
      listeSimpleInt.ajouter(32);
      listeSimpleInt.ajouter(-58);
      int x;
      for(int i=0; i<=1;i++)
      {
         x = listeSimpleInt. afficherInt (i);
         Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
   }
}
```

Qui affichera lors de l'exécution à la console :
listeSimpleInt : 32
listeSimpleInt : -58

Remarque n°1 :

> Dans la méthode afficherInt() l'instruction "**return** (**int**)**this**.liste[rang];". Nous sommes obligés de transtyper l'élément de rang "i" car dans l'ArrayList nous avons stocké des éléments de type object.

Remarque n°2 :

> Nous pouvons parfaitement ranger dans le même objet listeSimpleInt de type ListeSimple des **string** en construisant la méthode afficherStr() de la même manière que la méthode afficherInt( ) :
>
> **public string** afficherStr(**int** rang)

```
{
        return (string)this.liste[rang];
}
```

Soit le code de la classe ainsi construite :

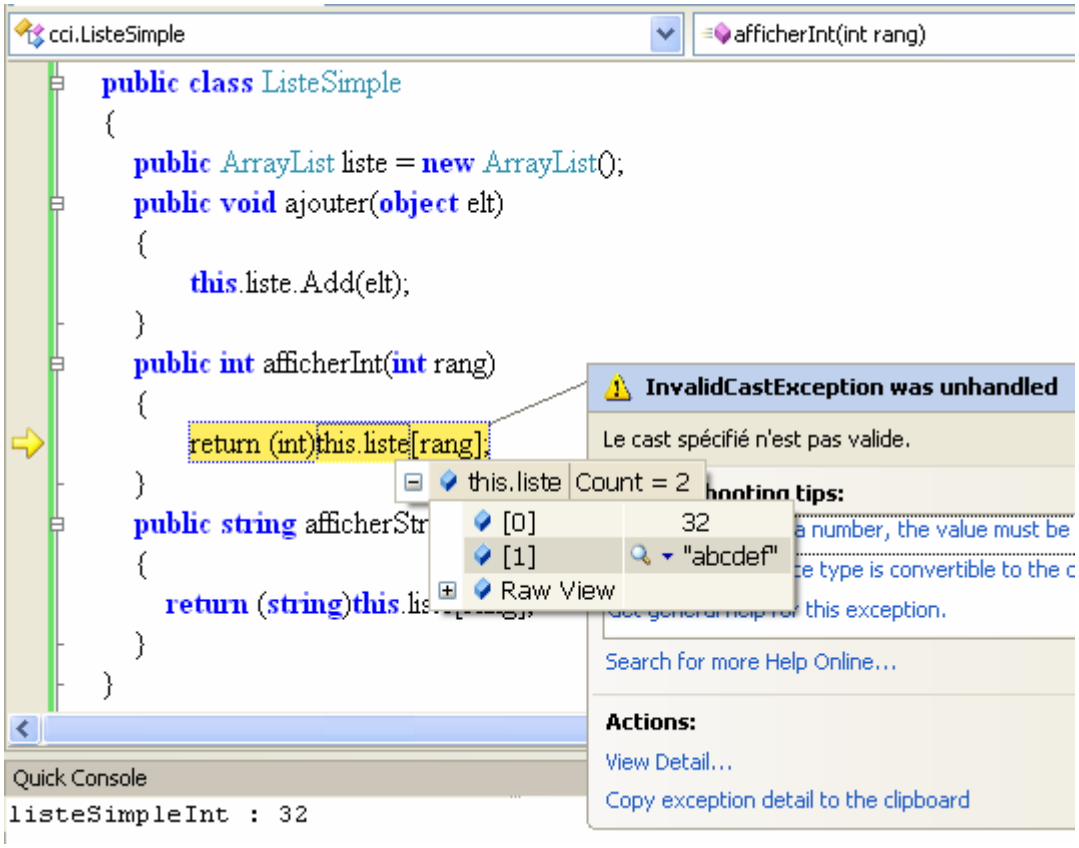
```
public class ListeSimple
{
    public ArrayList liste = new ArrayList();

    public void ajouter(object elt)
    {
        this.liste.Add(elt);
    }
    public int afficherInt(int rang)
    {
        return (int)this.liste[rang];
    }
    public string afficherStr(int rang)
    {
        return (string)this.liste[rang];
    }
}
```

Si nous compilons cette classe, le compilateur n'y verra aucune erreur, car le type **string** hérite du type object tout comme le type **int**. Mais alors, si par malheur nous oublions lors de l'écriture du code d'utilisation de la classe, de ne ranger que des éléments du même type (soit uniquement des **int**, soit uniquement des **string**) alors l'un des transtypage produira une erreur.

Le programme ci-dessous range dans l'ArrayList un premier élément de type **int**, puis un second élément de type **string**, et demande l'affichage de ces deux éléments par la méthode afficherInt( ). Aucune erreur n'est signalée à la compilation alors qu'il y a incompatibilité entre les types. Ceci est normal puisque le type de l'objet est obtenu dynamiquement :

```
class Program
  {
    static void Main(string[] args)
    {
      ListeSimple listeSimpleInt = new ListeSimple();
      listeSimpleInt.ajouter ( 32 );
      listeSimpleInt.ajouter ( "abcdef" );
      int x;
      for(int i=0; i<=1;i++)
      {
        x = listeSimpleInt. afficherInt (i);
        Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
    }
  }
```

Le transtypage du deuxième élément qui est une string en un int, est une erreur qui est signalée par le CLR lors de l'exécution :



Il revient donc au développeur dans cette classe à prêter une attention de tous les instants sur les types dynamiques des objets lors de l'exécution et éventuellement de gérer les incompatibilités par des gestionnaires d'exception try…catch.

Nous allons voir dans le prochain paragraphe comment les generics améliorent la programmation de ce problème.

1.2 Une liste générale avec les generics

Nous construisons une classe de type T générique agrégeant une liste d'éléments de type T générique. Dans ce contexte il n'est plus nécessaire de transtyper, ni de construire autant de méthodes que de types différents à stocker.

```
public class ListeGenerique <T>
 {
    List<T> liste = new List<T>();
    public void ajouter ( T elt )
    {
       this.liste.Add(elt);
    }
    public T afficher(int rang)
    {
       return this.liste[rang];
    }
 }
```

C'est dans le code source et donc lors de la compilation que le développeur définit le type <T> de l'élément et c'est le compilateur qui vérifie que chaque élément ajouté est bien de type <T> ainsi que chaque élément affiché est de type <T> :

L'instruction qui suit permet d'instancier à partir de la classe ListeGenerique un objet listeGenricInt d'éléments de type **int** par exemple :

```
ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
```

On peut instancier à partir de la même classe ListeGenerique un autre objet listeGenricStr d'éléments de type **string** par exemple :

```
ListeGenerique< string > listeGenricStr = new ListeGenerique< string >();
```

Le compilateur refusera le mélange des types :

```
class Program
  {
    static void Main(string[] args)
    {
      ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
      listeGenricInt.ajouter ( 32 );
      listeGenricInt.ajouter ( "abcdef" ); ← erreur signalée par le compilateur ici !
      int x;
      for(int i=0; i<=1;i++)
      {
        x = listeGenricInt.afficher (i);
        Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
    }
  }
```

Un programme correct serait :

```
class Program
  {
    static void Main(string[] args)
    {
      ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
      listeGenricInt.ajouter ( 32 );
      listeGenricInt.ajouter ( -58 );
      int x;
      for(int i=0; i<=1;i++)
      {
        x = listeGenricInt.afficher (i);
        Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
    }
  }
```

On procéderait d'une manière identique avec un objet "ListeGenerique< **string** > listeGenricStr = **new** ListeGenerique< **string** >()" dans lequel le compilateur n'acceptera que l'ajout d'éléments de type **string**.

1.3 Une méthode avec un paramètre générique

Nous reprenons la construction précédente d'une liste générique :

```
public class ListeGenerique <T>
{
   public List<T> liste = new List<T>();
   public void ajouter ( T elt )
   {
      this.liste.Add(elt);
   }
   public T afficher(int rang)
   {
      return this.liste[rang];
   }
}
```

Nous ajoutons dans la classe Program une méthode générique "**static void** afficher<T>(ListeGenerique<T> objet, **int** rang)" qui reçoit en paramètre formel une liste générique nommée objet de type <T>, nous la chargeons d'afficher l'élément de rang k de la liste et le type dynamique de l'objet obtenu par la méthode GetType(). L'appel de la méthode afficher nécessite deux informations :

a) le type paramétré de définition de l'objet passé en paramètre effectif, soit ici < **int** > ,
b) le paramètre effectif lui-même soit ici listeGenricInt.

```
class Program
  {
    static void afficher<T>(ListeGenerique<T> objet, int k)
    {
      Console.WriteLine(objet.liste[k] + ", de type : " + objet.liste[k].GetType());
    }
     static void Main(string[] args)
    {
      ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
      listeGenricInt.ajouter ( 32 );
      listeGenricInt.ajouter ( -58 );
      afficher <int> ( listeGenricInt, 0 );
      Console.WriteLine();
    }
  }
```

Résultat obtenu lors de l'exécution sur la console :
32, de type : System.Int32

1.4 Type générique contraint

Il est possible de contraindre un type paramétré à hériter d'une ou plusieurs classes génériques ou non et à implémenter une ou plusieurs interfaces classes génériques ou non en utilisant le mot clef where. Cette contrainte apporte deux avantages au développeur :

1°) améliorer la sécurité de vérification du typage lors de la compilation,
2°) réduire le transtypage.

Les options de contraintes d'un type paramétré sont les suivantes :

Syntaxe de la contrainte	Signification de la contrainte
where T: **struct**	Le type T doit être un type valeur.
where T : **class**	Le type T doit être un type référence.
where T : **new**()	Le type T doit avoir un constructeur sans paramètre explicite.
where T : < classeType>	Le type T doit hériter de la classe classeType
where T : <interfaceType>	Le type T doit implémenter l'interface interfaceType

Le(s) mot(s) clef **where** doit se situer avant le corps de la classe ou de la méthode.

Syntaxe d'utilisation par l'exemple dans une classe :

```
class clA {   ....  }
interface IGeneric1<T>  {   ....  }

class ClassGeneric<UnType, Autre>
    where UnType : clA, new()
    where Autre : class, IGeneric<UnType>
 {
    ......
 }
```

Dans l'exemple précédent la classe ClassGeneric est paramétrée par les deux types UnType et Autre qui supportent chacun une série de contraintes :

- Le type UnType est contraint d'hériter de la classe clA et doit avoir un constructeur sans paramètre explicite.
- Le type Autre est contraint d'être un type référence et doit implémenter l'interface IGeneric avec comme type paramétré <UnType>.

Syntaxe d'utilisation par l'exemple dans une méthode :

```
 public T meth1 < T, U >( )
        where T : new( )
        where U : ClassGeneric <T>
 {
        return new T( );
 }
```

Dans l'exemple précédent la méthode meth1 est paramétrée par les deux types T et U qui supportent chacun une série de contraintes (elle renvoie un résultat de type T) :

- Le type T est contraint d'avoir un constructeur sans paramètre explicite.
- Le type U est contraint d'hériter de la classe ClassGeneric avec <T> comme type paramétré.

1.5 Surcharge générique d'une classe

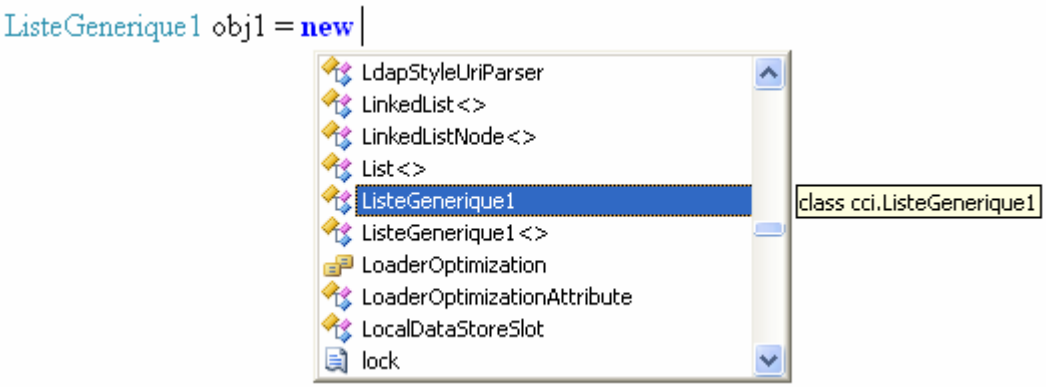
Les types paramétrés font partie intégrante de l'identification d'une classe aussi bien que le nom de la classe elle-même. A l'instar des méthodes qui peuvent être surchargées, il est possible pour une même classe de disposer de plusieurs surcharges : même nom, types paramètres différents.

Voici par exemple **trois classes distinctes** pour le compilateur C#, chacune de ces classes n'a aucun rapport avec l'autre si ce n'est qu'elle porte le même nom et qu'elles correspondent chacune à une surcharge différente de la classe ListeGenerique1 :

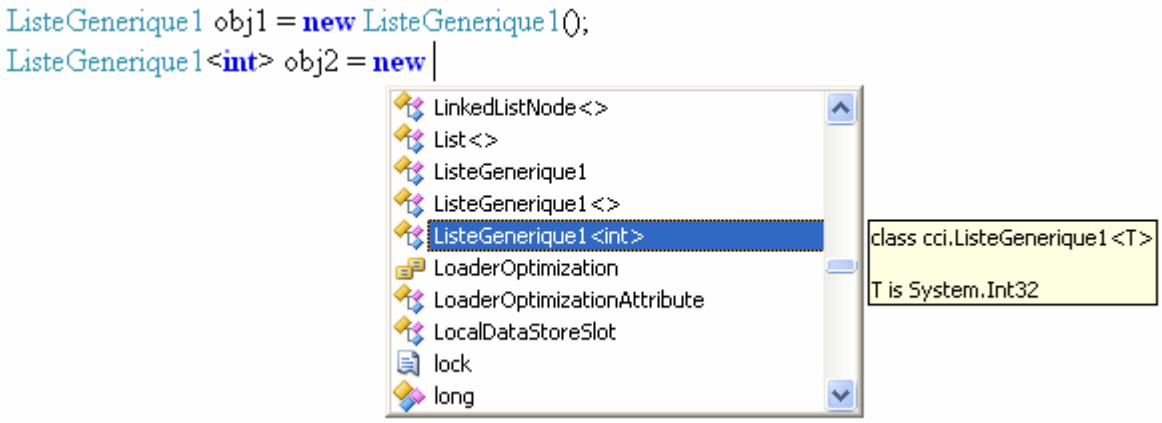
```
public class ListeGenerique1<T>
{
}
public class ListeGenerique1
{
}
public class ListeGenerique1<T, U>
{
}
```

Cet aspect comporte pour le développeur non habitué à la surcharge de classe à un défaut apparent de lisibilité largement compensé par l'audit de code dans Visual C#.

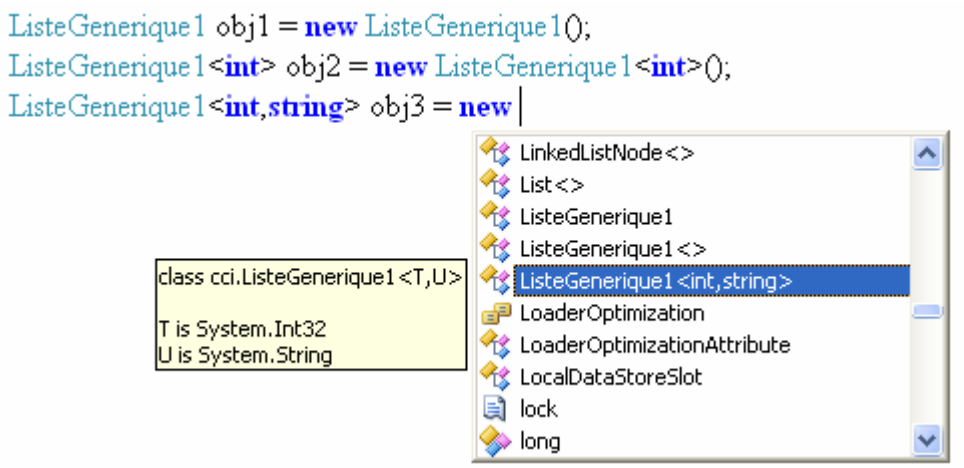
Exemple d'instanciation sur la première surcharge générique :



Exemple d'instanciation sur la seconde surcharge générique :



Exemple d'instanciation sur la troisième surcharge générique :



Les trois instanciations obtenues créent trois objets obj1 , obj2 , obj3 différents et de classe différente :

```
ListeGenerique1 obj1 = new ListeGenerique1();
ListeGenerique1<int> obj2 = new ListeGenerique1<int>();
ListeGenerique1<int, string> obj3 = new ListeGenerique1<int, string>();
```

On peut définir une surcharge générique d'une classe à partir d'une **autre surcharge générique de la même classe** :

```
public class ListeGenerique1<T, U>
{
}
public class ListeGenerique1<T> : ListeGenerique1<T, int>
{
}
```

Il est bien entendu possible de définir des surcharges génériques d'une classe à partir d'**autres surcharges génériques d'une autre classe** :

```
public class ListeGenerique1<T, U>    { … }

public class ListeGenerique1<T> : ListeGenerique1<T, int> { … }

public class ListeGenerique2 <U>: ListeGenerique1<string, U> { … }

public class ListeGenerique2 : ListeGenerique1<string, int> { … }

public class ListeGenerique2<T,U> : ListeGenerique1<T> { … }
```

Les types partiels

Plan général:

1. Les types partiels

1.1 Déclaration de classe partielle
1.2 classe partielle : héritage, implémentation et imbrication
1.3 classe partielle : type générique et contraintes

1. types partiels

Depuis la version 2.0, C# accepte la définition de struct, de classe, d'interface séparées. Ce qui revient à pouvoir définir à plusieurs endroits distincts dans un même fichier, un des trois types précédents ou encore le même type peut voir sa définition répartie sur plusieurs fichiers séparés. Cette notion de classe partielle est largement utilisée depuis 2.0 par Visual C# avec les WinForms.

Pour permettre ce découpage en plusieurs morceaux d'un même type, il faut obligatoirement utiliser dans la déclaration et juste avant la caractérisation du type (**struct**, **class**, **interface**), le modificateur **partial**.

Syntaxe:

```
partial class Truc1 { ….. }
partial struct Truc2 { ….. }
partial interface Truc3 { ….. }
```

Les autres modificateurs définissant les qualités du type (**static**, **public**, **internal, sealed, abstract**) doivent se trouver avant le mot clef **partial** :

```
static public partial class Truc1 { ….. }
internal partial struct Truc2 { ….. }
public partial interface Truc3 { ….. }
```

1.1 Déclaration de classe partielle

Nous étudions dans ce paragraphe, les implications de la déclaration d'une classe partielle sur son utilisation dans un programme.

soit un fichier *partie1.cs* contenant la classe ClassPartielle possédant un attribut entier public y initialisé à 200 dans l'espace de nom **cci** :

```
namespace cci
{
    partial class ClassPartielle
    {
        public int y = 200;
    }
}
```

On peut définir dans un autre fichier *partie2.cs* la "suite" de la classe ClassPartielle avec un autre attribut **public** x initialisé à 100 dans l'espace de même nom **cci** :

```
namespace cci
{
    partial class ClassPartielle
    {
        public int x = 100;
    }
}
```

Ajoutons dans le code du second fichier *partie2.cs*, une classe Program contenant la méthode **Main**, l'audit de code de C#, nous montre bien qu'un objet de type ClassPartielle, possède bien les deux attributs x et y :

```
namespace cci
{
    partial class ClassPartielle
    {
        public int x = 100;
    }
    class Program
    {
        static void Main(string[] args)
        {
            ClassPartielle Obj = new ClassPartielle();
            Obj.|        }
    }
}
```

Equals
GetHashCode
GetType
ToString
x
y
int ClassPartielle.y

3 déclarations partielles de la même classe	Implications sur la classe

partial class ClassPartielle {
  public int x = 100;
}
….
sealed partial class ClassPartielle {
  public int y = 200;
}
….
internal partial class ClassPartielle {
  public int z = 300;
}
	La classe est considérée par le compilateur comme possédant la réunion des deux qualificateurs **sealed** et **internal** :
	
	sealed internal partial class ClassPartielle
	{
	  ….
	}

Attention le compilateur C# vérifie la cohérence entre tous les déclarations différentes des qualificateurs d'une même classe partielle. Il décèlera par exemple une erreur dans les

déclarations suivantes :

```
public partial class ClassPartielle {
      public int x = 100;
}
….
internal partial class ClassPartielle {
       public int y = 200;
}
```

Dans l'exemple ci-dessus, nous aurons le message d'erreur du compilateur C# suivant :
Error : Les déclarations partielles de 'ClassPartielle' ont des modificateurs d'accessibilité en conflit.

Conseil : pour maintenir une bonne lisibilité du programme mettez tous les qualificateurs devant chacune des déclarations de classe partielle.

```
sealed internal partial class ClassPartielle {
      public int x = 100;
}
….
sealed internal partial class ClassPartielle {
       public int y =
}
….etc
```

Conseil : ne pas abuser du concept partial, car bien que pratique, la dissémination importante des membres d'une classe dans plusieurs fichiers peut nuire à la lisibilité du programme !

1.2 classe partielle : héritage, implémentation et imbrication

Héritage de classe

Si une classe partielle ClassPartielle hérite dans l'une de ses définitions d'une autre classe ClasseA partielle ou non, cet héritage s'étend implicitement à toutes les autres définitions de ClassPartielle :

```
public class ClasseA {
….
}
public partial class ClassPartielle : ClasseA {
      public int x = 100;
}….
public partial class ClassPartielle {
       public int y = 200;
}….
public partial class ClassPartielle {
      public int z = 300;
}….
```

Conseil : pour maintenir une bonne lisibilité du programme mettez la qualification d'héritage dans chacune des déclarations de la classe partielle.

```
public class ClasseA {
....
}
public partial class ClassPartielle : ClasseA {
      public int x = 100;
}
public partial class ClassPartielle : ClasseA {
       public int y = 200;
}
public partial class ClassPartielle : ClasseA {
      public int z = 300;
}
```

Implémentation d'interface

Pour une classe partielle implémentant une ou plusieurs interfaces le comportement est identique à celui d'une classe partielle héritant d'une autre classe. Vous pouvez écrire :

```
public interface InterfA { … }
public interface InterfB { … }

public partial class ClassPartielle : InterfA {
      public int x = 100;
}....
public partial class ClassPartielle : {
       public int y = 200;
}....
public partial class ClassPartielle : InterfB {
      public int z = 300;
}....
```

Ecriture plus lisible conseillée :

```
public interface InterfA { … }
public interface InterfB { … }

public partial class ClassPartielle : InterfA , InterfB {
      public int x = 100;
}....
public partial class ClassPartielle : InterfA , InterfB {
       public int y = 200;
}....
public partial class ClassPartielle : InterfA , InterfB {
      public int z = 300;
}....
```

Classe imbriquée

Les classes imbriquées de C# peuvent être définies sous forme de classes partielles.

Exemple d'imbrication dans une classe non partielle :

```
class classeMere
{
      partial class classeA
```

```
    {
    }
     …..
    partial class classeA
    {
    }
}
```

imbrication dans une classe partielle :

```
partial class classeMere
{
    partial class classeA
    {
    }
     …..
    partial class classeA
    {
    }
}
 ……
partial class classeMere
{
    partial class classeA
    {
    }
     …..
    partial class classeA
    {
    }
}
```

1.3 classe partielle : type générique et contraintes

Une classe générique peut être partielle

soit la classe de liste générique définie plus haut mise sous forme de 3 définitions partielles :

Première définition partielle :

```
public partial class ListeGenerique <T>
{
    List<T> liste = new List<T>();
}
```

Seconde définition partielle :

```
public partial class ListeGenerique <T>
{
    public void ajouter ( T elt )
    {
       this.liste.Add(elt);
    }
}
```

Troisième définition partielle :

```
public partial class ListeGenerique <T>
{
    public T afficher(int rang)
    {
      return this.liste[rang];
    }
}
```

Notons que le ou les types génériques paramétrant la classe partielle doivent obligatoirement être présents, sauf à respecter les conventions de la surcharge générique de classe vue plus haut.

Une classe partielle peut avoir des surcharges génériques partielles

Dans ce cas aussi nous ne saurions que trop conseiller au lecteur d'écrire explicitement dans toutes les définitions partielles d'une même surcharge, les mêmes listes de types paramétrés de la définition.

Notre exemple ci-dessous montre trois surcharges génériques d'une classe ListeGenerique1 : soit ListeGenerique1, ListeGenerique1<T> et ListeGenerique1<T, U>, chaque surcharge générique est déclarée sous forme de deux définitions de classes partielle.

```
public partial class ListeGenerique1<T>
{
}
public partial class ListeGenerique1<T>
{
}
public partial class ListeGenerique1
{
}
public partial class ListeGenerique1
{
}
public partial class ListeGenerique1<T, U>
{
}
public partial class ListeGenerique1<T, U>
{
}
```

Types génériques contraints dans les classes partielles

Il semble inutile de jongler avec les exclusions possibles ou non sur la position des différents paramètres de la clause **where** dans plusieurs déclarations d'une même classe partielle, nous proposons de jouer la lisibilité en explicitant la (les) même(s) clause(s) **where** dans chaque déclaration de la classe partielle :

Première déclaration partielle :

```
public class ClasseA { ….}
public interface InterfB { … }
```

```
public partial class ListeGenerique <T>
 where T : ClasseA , InterfB , new( )
{
      List<T> liste = new List<T>();
}
```

Seconde déclaration partielle :

```
public partial class ListeGenerique <T>
 where T : ClasseA , InterfB , new( )
{
      public void ajouter ( T elt )
      {
         this.liste.Add(elt);
      }
}
```

Troisième déclaration partielle :

```
public partial class ListeGenerique <T>
 where T : ClasseA , InterfB , new( )
{
      public T afficher(int rang)
      {
         return this.liste[rang];
      }
}
```

Les méthodes anonymes

Plan général:

1. Les méthodes anonymes

1.1 délégué et méthode anonyme
1.2 Création pas à pas d'une méthode anonyme
1.3 Gestionnaire anonyme d'événement classique sans information
1.4 Gestionnaire anonyme d'événement personnalisé avec information
1.5 Les variables capturées par une méthode anonyme
1.6 Les méthodes anonymes sont implicitement de classe ou d'instance
1.7 Comment les méthodes anonymes communiquent entre elles

1. Les méthodes anonymes

Le concept de méthode anonyme dans C# est semblable au concept de classe anonyme en Java très utilisé pour les écouteurs.

L'objectif est aussi dans C# de réduire les lignes de code superflues tout en restant lisible. Les méthodes anonymes en C# sont intimement liées aux délégués.

1.1 Délégué et méthode anonyme

Là où le développeur peut mettre un objet délégué, il peut aussi bien mettre une méthode anonyme. Lorsque dans le programme, il n'est pas nécessaire de connaître le nom de la méthode vers laquelle pointe un délégué, les méthodes anonymes sont alors un bon outil de simplification du code.

Prenons un exemple d'utilisation classique d'un délégué censé permettre fictivement de pointer vers des méthodes de recherche d'un nom dans une liste. Rappelons la démarche générale pour la manipulation de ce concept :

```
 // 1°) la classe de délégué
delegate string DelegListe(int rang);

 class ClassMethodeAnonyme
 {
     // 2°) la méthode vers laquelle va pointer le délégué
    private string searchNom1(int x)
    {
        return "found n° "+Convert.ToString(x);
    }
    public void utilise()
    {
        // 3°) création d'un objet délégué pointant vers la méthode
        DelegListe search1 = new DelegListe(searchNom1);
        Console.WriteLine(search1(1));
    }
 }
```

Reprenons les mêmes éléments mais avec une méthode anonyme :

```
// 1°) la classe de délégué
 delegate string DelegListe(int rang);

 class ClassMethodeAnonyme
 {
     // 2°) la méthode anonyme vers laquelle pointe le délégué
    private DelegListe searchNom2 = delegate ( int x )
    {
        return "found n° "+Convert.ToString(x);
    }
```

```
    public void utilise()
    {
        Console.WriteLine(searchNom2 (2));
    }
}
```

1.2 Création pas à pas d'une méthode anonyme

Explicitons la démarche ayant conduit à l'écriture du remplacement dans le premier paragraphe précédent des étapes 2° et 3° par la seule étape 2° avec une méthode anonyme dans le second paragraphe.

1°) Une méthode anonyme est déclarée par le mot clef général **delegate** avec son corps de méthode complet :

```
    delegate ( int x )
    {
        return "found n° "+Convert.ToString(x);
    }
```

2°) On utilise une référence de **delegate** , ici searchNom2 de type DelegListe :

```
    private DelegListe searchNom2
```

3°) Une méthode anonyme possède la signature de la classe **delegate** avec laquelle on souhaite l'utiliser (ici DelegListe) directement par l'affectation à une référence de **delegate** appropriée :

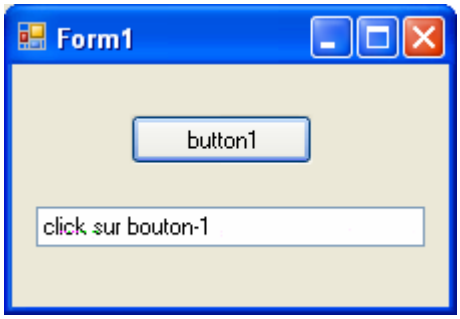
```
    private DelegListe searchNom2 = delegate ( int x )
    {
        return "found n° "+Convert.ToString(x);
    }
```

Si le lecteur souhaite exécuter l'exemple complet, voici le code de la méthode Main de la classe principale affichant les résultats fictifs :

```
class Program
{
    static void Main(string[] args)
    {
        ClassMethodeAnonyme obj = new ClassMethodeAnonyme();
        obj.utilise();
        Console.ReadLine();
    }
}
```

1.3 Gestionnaire anonyme d'événement classique sans information

Reprenons un exemple de construction et d'utilisation d'un **événement classique sans information** comme le click sur le bouton dans la fenêtre ci-dessous, et utilisons un gestionnaire d'événements anonyme.



La démarche classique (prise en charge automatiquement dans les environnements Visual Studio, Borland Studio, sharpDevelop) est la suivante :

- Créer et faire pointer l'événement Click (qui est un **delegate**) vers un nom de gestionnaire déjà défini (ici button1_Click) :

  ```
  this.button1.Click += new System.EventHandler(this.button1_Click);
  ```

- Définir le gestionnaire button1_Click selon la signature du **delegate** :

  ```
  private void button1_Click(object sender, EventArgs e)
  {
          textBox1.Text = "click sur bouton-1";
  }
  ```

L'utilisation d'un **gestionnaire anonyme** réduit ces deux instructions à une seule et au même endroit :

```
this.button1.Click += delegate (object sender, System.EventArgs e)
{
          textBox1.Text = "click sur bouton-1";
};
```

1.4 Gestionnaire anonyme d'événement personnalisé avec information

Soit un exemple d'utilisation de 4 gestionnaires anonymes d'un même événement personnalisé avec information appelé **Enlever**. Pour la démarche détaillée de création d'un tel événement que nous appliquons ici, nous renvoyons le lecteur au chapitre "Evénements" de cet ouvrage.

Nous déclarons une classe d'informations sur un événement personnalisé :

```
public class EnleverEventArgs : EventArgs { … }
```

Nous déclarons la classe delegate de l'événement personnalisé :

```
public delegate void DelegueEnleverEventHandler (object sender,
EnleverEventArgs e);
```

Nous donnons le reste sans explication supplémentaire autre que les commentaires inclus dans le code source,

```
using System;
using System.Collections;

namespace cci {

   //--> 1°) classe d'informations personnalisées sur l'événement
   public class EnleverEventArgs : EventArgs
   {
      public string info;
      public EnleverEventArgs(string s)
      {
         info = s;
      }
   }
   //--> 2°) déclaration du type délégation normalisé
   public delegate void DelegueEnleverEventHandler(object sender, EnleverEventArgs e);

   public class ClassA
   {
      //--> 3°) déclaration d'une référence event de type délégué :
      public event DelegueEnleverEventHandler Enlever;
      //--> 4.1°) méthode protégée qui déclenche l'événement :
      protected virtual void OnEnlever(object sender, EnleverEventArgs e)
      {
         //....
         if (Enlever != null) Enlever(sender, e);
         //....
      }
      //--> 4.2°) méthode publique qui lance l'événement :
      public void LancerEnlever()
      {
         //....
         EnleverEventArgs evt = new EnleverEventArgs("événement déclenché");
         OnEnlever(this, evt);
         //....
      }
   }

   public class ClasseUse
   {
      //--> 5°) la méthode permettant l'utilisation des gestionnaires anonymes
      static public void methodUse()
      {
         ClassA ObjA = new ClassA();
         ClasseUse ObjUse = new ClasseUse();

         //--> 6°) abonnement et définition des 4 gestionnaires anonymes :
         ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
         {
            //...gestionnaire d'événement Enlever: méthode d'instance.
            System.Console.WriteLine("information utilisateur 100 : " + e.info);
         };
```

```
        ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
        {
           //...gestionnaire d'événement Enlever: méthode d'instance.
           System.Console.WriteLine("information utilisateur 101 : " + e.info);
        };
        ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
        {
           //...gestionnaire d'événement Enlever: méthode d'instance.
           System.Console.WriteLine("information utilisateur 102 : " + e.info);
        };
        ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
        {
           //...gestionnaire d'événement Enlever: méthode de classe.
           System.Console.WriteLine("information utilisateur 103 : " + e.info);
        };

        //--> 7°) consommation de l'événement:
        ObjA.LancerEnlever();  //...l'appel à cette méthode permet d'invoquer l'événement Enlever
     }
     static void Main(string[] args)
     {
        ClasseUse.methodUse();
        Console.ReadLine();
     }
   }
}
```

Résultats d'exécution du programme précédent avec gestionnaires anonymes :

```
D:\CsBuilder\EventPerso\bin\Debug\ProjEventPerso.exe
information utilisateur 100 : événement déclenché
information utilisateur 101 : événement déclenché
information utilisateur 102 : événement déclenché
information utilisateur 103 : événement déclenché
```

1.5 Les variables capturées par une méthode anonyme

Une méthode anonyme accède aux données locales du bloc englobant dans lequel elle est définie.

- Les variables locales utilisables par une méthode anonyme sont appelées **variables externes**.
- Les variables utilisées par une méthode anonyme sont appelées les **variables externes capturées**.

Exemple :

```
int y =100; // variable externe au bloc anonyme

.... = delegate (int x)
      {
         y = x-3; // variable externe x capturée par le bloc anonyme
      };
....
```

Tous type de variables (valeur ou référence) peut être capturé par un bloc anonyme.

Le bloc englobant peut être une méthode :

```
delegate string DelegateListe(int rang);
…….
public void utilise ( )
{
    int entier = 100; //...variable locale type valeur
    object Obj = null; //... variable locale type référence

  DelegateListe searchNum =  delegate(int x)
  {
     entier = 99; //...variable capturée
     Obj = new object ( ); //...variable capturée
     return "found n° " + Convert.ToString(x);
  };
  Console.WriteLine("static : " + searchNum(2));
}
```

Le bloc englobant peut être une classe :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //...champ de classe type valeur
   int entierInstance = 200;  //...champ d'instance type valeur
   object Obj = null; //... champ d'instance type référence

   public void utiliseMembre ( )
   {
      DelegateListe searchNum =  delegate(int x)
      {
         entierStatic++; //...variable capturée
         entierInstance++;  //...variable capturée
         Obj = new object ( ); //...variable capturée
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(1) );
  }
```

Il est bien sûr possible de combiner les deux genres de variables capturées, soit local, soit membre.

1.6 Les méthode anonymes sont implicitement de classe ou d'instance

Une méthode anonyme peut être implicitement de classe ou bien implicitement d'instance **uniquement**, selon que le délégué qui pointe vers la méthode est lui-même dans une méthode de classe ou une méthode d'instance.

Le fait que le délégué qui pointe vers une méthode anonyme soit explicitement **static**, n'induit pas que la méthode anonyme soit de classe; c'est la méthode englobant la méthode anonyme qui impose son modèle **static** ou d'instance.

a) Exemple avec **méthode englobante d'instance** et **membre délégué d'instance** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   DelegateListe searchNum; //...membre délégué d'instance

   public void utiliseMembre ( )
   {
      searchNum =  delegate(int x)
      {
         entierStatic++;//...variable de classe capturée
         entierInstance++; //...variable d'instance capturée
         Obj = new object ( ); //...variable d'instance capturée
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(1) );
  }
}
```

b) Exemple avec **méthode englobante d'instance** et **membre délégué de classe** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   static DelegateListe searchNum; //...membre délégué de classe

   public void utiliseMembre ( )
   {
      searchNum =  delegate(int x)
      {
         entierStatic++;//...variable de classe capturée
         entierInstance++; //...variable d'instance capturée
         Obj = new object ( ); //...variable d'instance capturée
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(2) );
  }
}
```

c) Exemple avec **méthode englobante de classe** et **membre délégué de classe** explicite :

```
delegate string DelegateListe(int rang);
```

```
class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   static DelegateListe searchNum; //...membre délégué de classe

   public static void utiliseMembre ( )
   {
      searchNum =  delegate(int x)
      {
         entierStatic++; //...variable static capturée
         entierInstance++; ← erreur de compilation membre d'instance non autorisé
         Obj = new object ( ); ← erreur de compilation membre d'instance non autorisé
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(3) );
   }
}
```

d) Exemple avec **méthode englobante de classe** et **membre délégué d'instance** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   DelegateListe searchNum; //...membre délégué d'instance

   public static void utiliseMembre ( )
   {
      searchNum =  delegate(int x) ← erreur de compilation le  membre doit être static
      {
         entierStatic++; //...variable static capturée
         entierInstance++; ← erreur de compilation membre d'instance non autorisé
         Obj = new object ( ); ← erreur de compilation membre d'instance non autorisé
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(4) );
   }
}
```

Nous remarquons dans les exemples précédents, que le délégué **searchNum** qui pointe vers la méthode anonyme **delegate(int** x) { … } possède bien les mêmes caractéristiques que la méthode englobante **utiliseMembre ( ).** C'est pourquoi dans les exemples (c) et (d) **searchNum** est implicitement **static** comme **utiliseMembre( )** et donc les erreurs signalées par le compilateur dans la méthode anonyme, sont des erreurs classiques commises sur une méthode qui est **static** et qui ne peut accéder qu'à des entités elles-mêmes **static**.

1.7 Comment les méthode anonymes communiquent entre elles

La notion de variable externe capturée par une méthode anonyme permet à plusieurs méthodes anonymes déclarées dans le même bloc englobant de partager et donc capturer les mêmes variables externes.

Ces variables sont donc comme des variables communes à toutes les méthodes anonymes qui peuvent ainsi communiquer entre elles comme dans l'exemple ci-dessous où deux méthodes anonymes pointées l'une par le délégué **searchNum**, l'autre par le délégué **combienDeSearch** partagent les mêmes variables **numero** (variable locale au bloc englobant) et **nbrDeRecherche** membre d'instance de la classe :

```
class ClasseA
{
   int nbrDeRecherche = 0;  //... membre d'instance type valeur
   DelegateListe searchNum; //...membre délégué d'instance
   DelegateConsulter combienDeSearch; //...membre délégué d'instance

   public void utiliseMembre()
   {
      int numero = 12345;
      searchNum = delegate(int x)
      {
         nbrDeRecherche++;//...variable d'instance capturée
         numero++; //...variable locale capturée
         return "found n° " + Convert.ToString(x);
      };
      combienDeSearch = delegate()
      {
         return "nombre de recherches effectuées : " + Convert.ToString(nbrDeRecherche)
                  + " , numéro : " + Convert.ToString(numero);
      };
      Console.WriteLine( searchNum(50) );
      Console.WriteLine( combienDeSearch( ) );
   }
}
```

Résultats obtenu lors de l'exécution sur la console :
found n° 50
nombre de recherches effectuées : 1 , numéro : 12346

Attention : une méthode anonyme ne s'exécute que lorsque le délégué qui pointe vers elle est lui-même invoqué, donc les actions effectuées sur des variables capturées ne sont effectives que lors de l'invocation du délégué.

Si nous reprenons la classe précédente et que nous reportions l'initialisation "**numero = 12345;"**de la variable locale **numero** dans la première méthode anonyme, nous aurons un message d'erreur du compilateur :

```
class ClasseA
{
   int nbrDeRecherche = 0;  //... membre d'instance type valeur
   DelegateListe searchNum; //...membre délégué d'instance
   DelegateConsulter combienDeSearch; //...membre délégué d'instance
```

```
    public void utiliseMembre()
    {
        int numero; //...variable locale non initialisée
        searchNum = delegate(int x)
        {
            nbrDeRecherche++;//...variable d'instance capturée
            numero= 12345; //...variable locale capturée et initialisée
            return "found n° " + Convert.ToString(x);
        };
        combienDeSearch = delegate()
        {
            return "nombre de recherches effectuées : " + Convert.ToString(nbrDeRecherche)
                    + " , numéro : "
                    + Convert.ToString(numero); ← erreur de compilation variable numero non initialisée !
        };
        Console.WriteLine( searchNum(50) );
        Console.WriteLine( combienDeSearch( ) );
    }
}
```

Types abstraits de données , implantation avec classes génériques

Plan du chapitre:

1. Types abstraits de données et classes génériques en C#

1.1 Traduction générale TAD → classes
1.2 Exemples de Traduction TAD → classes
1.3 Variations sur les spécifications d'implantation
1.4 Exemples d'implantation de la liste linéaire
1.5 Exemples d'implantation de la pile LIFO
1.6 Exemples d'implantation de la file FIFO

1. Types abstraits de données et classes en C#

Dans cette partie nous adoptons un point de vue pratique dirigé par l'implémentation sous une forme accessible à un débutant des notions de type abstrait de donnée.

Nous allons proposer une écriture des TAD avec des classes C# :

- La notion classique de **classe**, contenue dans tout langage orienté objet, se situe au niveau 2 de cette hiérarchie **constitue une meilleure approche de la notion de TAD**.

En fait un TAD sera bien décrit par les membres **public** d'une **classe** et se traduit presque immédiatement ; le travail de traduction des préconditions est à la charge du développeur et se trouvera dans le corps des méthodes.

1.1 Traduction générale TAD → classe C#

Nous proposons un tableau de correspondance pratique entre la signature d'un TAD et les membres d'une classe :

syntaxe du TAD	*squelette de la classe associée*
TAD Truc	**class Truc {**
utilise TAD_1 , TAD_2,...,TAD_n	**Les classes** TAD_1,...,TAD_n sont déclarées dans le même espace de nom que truc (sinon le global par défaut)
champs **.....**	**attributs** **.....**
opérations Op1 : E x F → G Op2 : E x F x G → H x S	**public** G Op1(E x, F y) {} **public void** Op2(E x, F y, G, **ref** H t, **ref** S u) { } **.....**
FinTAD-Truc	**}**

Reste à la charge du programmeur l'écriture du code dans le corps des méthodes Op1 et Op2

1.2 Exemples de Traduction TAD → classe C#

Le TAD Booléens implanté sous deux spécifications concrètes en C# avec deux types scalaires différents.

Spécification opérationnelle concrète n°1

Les constantes du type Vrai, Faux sont représentées par deux attributs de type entier dans un type structure nommé « logique » :

```
public struct logique
{
    static public int Faux = 0;
    static public int Vrai = 1;
}
```

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens
Champs :
Opérations :
Vrai : → Booléens
Faux : → Booléens
Et : Booléens x Booléens → Booléens
Ou : Booléens x Booléens → Booléens
Non : Booléens → Booléens
 	
FINTAD-Booléens
	class Booleens
	{
	public logique val;
	
	public Booleens Et (Booleens x, Booleens y) { }
	public Booleens Ou (Booleens x, Booleens y) { }
	public Booleens Non (Booleens x) { }
	
	}

Spécification opérationnelle concrète n°2

Les constantes du type Vrai, Faux sont représentées par deux identificateurs C# dans un type énuméré nommé « logique » :

```
public enum logique { Faux, Vrai };
```

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens
Champs :
Opérations :
Vrai : → Booléens
Faux : → Booléens
Et : Booléens x Booléens → Booléens
Ou : Booléens x Booléens → Booléens
Non : Booléens → Booléens
 	
FINTAD-Booléens
	class Booleens
	{
	public logique val ;
	
	public Booleens Et (Booleens x, Booleens y) { }
	public Booleens Ou (Booleens x, Booleens y) { }
	public Booleens Non (Booleens x) { }
	
	}

Nous remarquons la forte similarité des deux spécifications concrètes :

Implantation avec des entiers	Implantation avec des énumérés

public struct logique
{
static public int Faux = 0;
static public int Vrai = 1;
}
	public enum logique = (faux , vrai) ;

class Booleens
{
public int val ;
public Booleens (**int** init)
{
val = init;
}
public Booleens Et (Booleens x, Booleens y) { }
public Booleens Ou (Booleens x, Booleens y) { }
public Booleens Non (Booleens x) { }
}
	class Booleens
	{
	public logique val ;
	public Booleens (logique init)
	{
	val = init;
	}
	public Booleens Et (Booleens x, Booleens y) { }
	public Booleens Ou (Booleens x, Booleens y) { }
	public Booleens Non (Booleens x) { }
	}

1.3 Variations sur les spécifications d'implantation

Cet exercice ayant été proposé à un groupe d'étudiants, nous avons eu plusieurs genres d'implantation des opérations : « et », « ou », « non ». Nous exposons au lecteur ceux qui nous ont parus être les plus significatifs :

Implantation d'après spécification concrète n°1

Fonction Et	Fonction Et

public Booleens Et (Booleens x, Booleens y)
{
return x.val * y.val ;
}
	public Booleens Et (Booleens x, Booleens y)
	{
	if (x.val == logique.Faux)
	return new Booleens (logique.Faux);
	else return new Booleens (y.val);
	
	}

Fonction Ou	Fonction Ou

public Booleens Ou (Booleens x, Booleens y)
{
return x.val +y.val - x.val *y.val ;
}
	public Booleens Ou (Booleens x, Booleens y)
	{
	if (x.val == logique.Vrai)
	return new Booleens (logique.Vrai);
	else return new Booleens (y.val);
	}

Fonction Non	Fonction Non

public Booleens Non (Booleens x)
{
return 1-x.val ;
}
	public Booleens Non (Booleens x)
	{
	if (x.val == logique.Vrai)
	return new Booleens (logique.Vrai);
	else return new Booleens (logique.Faux);
	}

Dans la colonne de gauche de chaque tableau, l'analyse des étudiants a été dirigée par le choix de la spécification concrète sur les entiers et sur un modèle semblable aux fonctions indicatrices des ensembles. Ils ont alors cherché des combinaisons simples d'opérateurs sur les entiers fournissant les valeurs adéquates.

Dans la colonne de droite de chaque tableau, l'analyse des étudiants a été dirigée dans ce cas par des considérations axiomatiques sur une algèbre de Boole. Ils se sont servis des propriétés d'absorbtion des éléments neutres de la loi " *ou* " et de la loi " *et* ". Il s'agit là d'une structure algébrique abstraite.

Influence de l'abstraction sur la réutilisation

A cette étape du travail nous avons demandé aux étudiants quel était, s'il y en avait un, le meilleur choix d'implantation quant à sa réutilisation pour l'implantation d'après la spécification concrète n°2.

Les étudiants ont compris que la version dirigée par les axiomes l'emportait sur la précédente, car sa qualité d'abstraction due à l'utilisation de l'axiomatique a permis de la réutiliser sans aucun changement dans la partie **implementation** de la unit associée à spécification concrète n°2 (en fait toute utilisation des axiomes d'algèbre de Boole produit la même efficacité).

Conclusion :

l'abstraction a permis ici une réutilisation totale et donc un gain de temps de programmation dans le cas où l'on souhaite changer quelle qu'en soit la raison, la spécification concrète.

1.4 Exemples d'implantation de liste linéaire générique

Rappel de l'écriture du TAD de liste linéaire

TAD Liste
utilise : **N**, T_0, Place
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
liste_vide : $\rightarrow$ Liste
acces : Liste x **N** $\rightarrow$ Place
contenu : Place $\rightarrow T_0$
kème : Liste x **N** $\rightarrow T_0$
long : Liste $\rightarrow$ **N**
supprimer : Liste x **N** $\rightarrow$ Liste
inserer : Liste x **N** x $T_0 \rightarrow$ Liste
ajouter : Liste x $T_0 \rightarrow$ Liste
succ : Place $\rightarrow$ Place

préconditions :
acces(L,k) **def_ssi** $1 \leq k \leq$ long(L)
supprimer(L,k) **def_ssi** $1 \leq k \leq$ long(L)
inserer(L,k,e) **def_ssi** $1 \leq k \leq$ long(L) +1
kème(L,k) **def_ssi** $1 \leq k \leq$ long(L)

Fin-Liste

Dans les exemples qui suivent, la notation ≅. indique la traduction en langage C#.

spécification proposée en C# :

Liste ≅	`interface IListe<T0> {`
	`int longueur`
	`{`
	`get;`
	`}`
	`T0 this[int index]`
	`{`
	`get;`
	`set;`
	`}`
	`bool est_Vide();`
	`void ajouter(T0 Elt);`
	`void inserer(int k, T0 x);`
	`void supprimer(int k);`
	`bool estPresent(T0 x);`
	`int rechercher(T0 x);`
	`}`
	`class Liste<T0> : IListe<T0> {`
	`public static readonly int max_elt = 100;`
	`private T0[] t;`
	`private int longLoc;`
	`public Liste()`
	`{`
	`t = new T0[max_elt];`
	`longLoc = 0;`
	`}`
	`// Le reste implante l'interface IListe<T0> …….....`
	`}`

liste_vide ≅	**Propriété** : longueur = 0
acces ≅	**Indexeur** : this[index]
contenu ≅	**Indexeur** : this[index]
kème ≅	**Indexeur** : this[index]
long ≅	**Propriété** : longueur
succ ≅	**Indexeur** : this[index]
supprimer ≅	`public void supprimer (int k) { ..... }`
inserer ≅	`public void inserer (int k , T0 Elt ) { ..... }`

La précondition de l'opérateur **supprimer** peut être ici implantée par le test :

```
if (0 <= k &&  k <= longueur - 1) …….
```

La précondition de l'opérateur **insérer** peut être ici implantée par le test :

```
if (longueur < max_elt && (0 <= k && k <= longueur) ) …….
```

Les deux objets **acces** et **contenu** ne seront pas utilisés en pratique, car l'indexeur de la classe les implante automatiquement d'une manière transparente pour le programmeur.

Le reste du programme est laissé au soin du lecteur qui pourra ainsi se construire à titre didactique, sur sa machine, une base de types en C# de base, il en trouvera une correction à la fin de ce chapitre.

Nous pouvons " **enrichir** " le TAD Liste en lui adjoignant deux opérateurs test et rechercher (rechercher un élément dans une liste). Ces adjonctions ne posent aucun problème. Il suffit pour cela de rajouter au TAD les lignes correspondantes :

opérations
estPresent : Liste x $T_0 \rightarrow$ Booléen
rechercher : Liste x $T_0 \rightarrow$ Place
précondition
rechercher(**L**,**e**) **def_ssi** Test(**L**,**e**) = **V**

Le lecteur construira à titre d'exercice l'implantation C# de ces deux nouveaux opérateurs en étendant le programme déjà construit. Il pourra par exemple se baser sur le schéma de représentation C# suivant :

```
public bool estPresent(T0 Elt)
{
 return !(rechercher(Elt) == -1);
}

public int rechercher(T0 Elt)
{
    // il s'agit de fournir le rang de x dans la liste
    // utiliser par exemple un algo de recherche séquentielle
}
```

1.5 Exemples d'implantation de pile LIFO générique

Rappel de l'écriture du TAD Pile LIFO

TAD PILIFO

utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
sommet : → PILIFO
Est_vide : PILIFO → Booléens
empiler : PILIFO x T_0 x sommet → PILIFO x sommet
dépiler : PILIFO x sommet → PILIFO x sommet x T_0
premier : PILIFO → T_0
préconditions :
dépiler(P) **def_ssi** est_vide(P) = **Faux**
premier(P) **def_ssi** est_vide(P) = **Faux**

FinTAD-PILIFO

Nous allons utiliser un **tableau** avec une case supplémentaire permettant d'indiquer que le fond de pile est atteint (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en C# :

Pilifo ≅	interface ILifo<T0>
	{
	int nbrElt
	{
	get;
	}
	bool estVide();
	void empiler(T0 Elt);
	T0 depiler();
	T0 premier();
	}
	
	class Lifo<T0> : ILifo<T0>
	{
	public static readonly int max_elt = 100;
	private T0[] t;
	private int sommet, fond;
	public Lifo()
	{
	t = new T0[max_elt];
	fond = -1;
	sommet = fond;
	// Le reste implante l'interface ILifo<T0> …….....
	}

depiler ≅	public T0 depiler ( )
empiler ≅	public T0 empiler (T_0 Elt)
premier ≅	public T0 premier ( )
Est_vide ≅	public bool estVide ( )

Le contenu des méthodes est conseillé au lecteur à titre d'exercice, il en trouvera une correction à la fin de ce chapitre..

Remarque :

Il est aussi possible de construire une spécification opérationnelle à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ". Il est vivement conseillé au lecteur d'écrire cet exercice en C# pour bien se convaincre de la différence entre les niveaux d'abstractions.

1.6 Exemples d'implantation de file FIFO générique

Rappel de l'écriture du TAD file FIFO

TAD FIFO
utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
  tête : → FIFO
  fin : → FIFO
  Est_vide : FIFO → Booléens
  ajouter : FIFO x T_0 x fin → PILIFO x fin
  retirer : FIFO x tête → FIFO x tête x T_0
  premier : FIFO → T_0
préconditions :
  retirer(F) **def_ssi** est_vide(F) = **Faux**
  premier(F) **def_ssi** est_vide(F) = **Faux**

FinTAD-FIFO

Nous allons utiliser ici aussi un **tableau** avec une case supplémentaire permettant d'indiquer que la file est vide (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en C# :

Fifo ≅

```
interface IFifo<T0> {
    int nbrElt
        { get;   }
    bool estVide();
    void ajouter(T0 Elt);
    T0 retirer();
    T0 premier();
 }

 class Fifo<T0> : IFifo<T0> {
    public static readonly int max_elt = 100;
    private T0[] t;
      private int tete, fin;
    public Fifo()
    {
       t = new T0[max_elt];
       tete = -1;
       fin = 0;
```

	} }
retirer ≅	public T0 retirer ( )
ajouter ≅	public T0 ajouter (T_0 Elt)
premier ≅	public T0 premier ( )
Est_vide ≅	public bool estVide ( )

Le contenu des méthodes est conseillé au lecteur à titre d'exercice, il en trouvera une correction à la fin de ce chapitre..

Remarque :

Comme pour le TAD Pilifo, il est aussi possible de construire une spécification opérationnelle du TAD FIFO à l'aide du TAD Liste en remplaçant dans l'étude précédente l'objet de tableau « private T0[] t » par l'objet de liste « private Liste<T_0> t » .

Une solution d'implantation de liste linéaire générique en C

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

 interface IListe<T0>
{
   int longueur
   {
      get;
   }
   T0 this[int index]
   {
      get;
      set;
   }
   bool est_Vide();
   void ajouter(T0 Elt);
   void inserer(int k, T0 x);
   void supprimer(int k);
   bool estPresent(T0 x);
   int rechercher(T0 x);
}
class Liste<T0> : IListe<T0>
{
   public static readonly int max_elt = 100;
   private T0[] t;
   private int longLoc;
   public Liste()
   {
      t = new T0[max_elt];
      longLoc = 0;
   }

   public void ajouter(T0 Elt)
   {
      if (longueur < max_elt)
      {
         t[longueur] = Elt;
         longueur++;
      }
      else
         System.Console.WriteLine("ajout impossible : capacité maximale atteinte !");
   }

   public bool est_Vide()
   {
      return longueur == 0;
   }

   public T0 this[int index]
   {
      get
      {
         if (!est_Vide())
         {
```

```
            if (0 <= index && index <= longueur)
            {
               return t[index];
            }
            else
            {
              System.Console.WriteLine("indexage, lecture incorrecte : indice (" + index + ") hors de la liste.");
              return default(T0);
            }
         }
         else
         {
            System.Console.WriteLine("indexage, lecture incorrecte : Désolé la liste est vide.");
            return default(T0);
         }
      }
      set {
         if (0 <= index && index <= longueur - 1)
         {
            t[index] = value;
         }
         else
            System.Console.WriteLine("indexage, écriture impossible : indice (" + index + ") hors de la liste.");
          }
   }

   public int longueur
   {
      get { return longLoc; }
      protected set { longLoc = value; }
   }

   public void supprimer(int k)
   {
      if (!est_Vide())
      {
         if (0 <= k && k <= longueur - 1)
         {
            for (int i = k; i < longueur - 1; i++)
               t[i] = t[i + 1];
            longueur--;
         }
         else
            System.Console.WriteLine("suppression impossible : indice (" + k + ") hors de la liste.");
      }
   }

   public void inserer(int k, T0 Elt)
   {
      if (!est_Vide())
      {
         if (longueur < max_elt && (0 <= k && k <= longueur))
         {
            for (int i = longueur - 1; i >= k; i--)
               t[i + 1] = t[i];
            t[k] = Elt;
            longueur++;
         }
         else
            if (longueur >= max_elt)
```

```
                System.Console.WriteLine("insertion impossible : capacité maximale atteinte !");
            else
                System.Console.WriteLine("insertion impossible : indice (" + k + ") hors de la liste.");
        }
        else
            ajouter(Elt);
    }

    public bool estPresent(T0 Elt)
    {
        return !(rechercher(Elt) == -1);
    }

    public int rechercher(T0 Elt)
    {
        int k;
        for (k = 0; k < longueur; k++)
            if (Elt.Equals(t[k])) break;
        if (k == longueur)
            return -1;
        else
            return k;
    }

    public void afficher()
    {
        for (int k = 0; k < longueur; k++)
            System.Console.Write(t[k] + ", ");
        System.Console.WriteLine(": long = " + longueur);
    }
}
```

Exemple de classe principale testant la classe « Liste » sur T_0 = **int** :

```
class ProgramListe
{
    static void Main(string[] args)
    {
        Liste<int> liste = new Liste<int>();
        System.Console.WriteLine(liste[0]);
        for (int i = 0; i < Liste<int>.max_elt / 10; i++)
            liste.inserer(i, i);
        liste.afficher();
        liste.inserer(20, 97);
        liste.inserer(liste.longueur, 98);
        liste.supprimer(liste.longueur - 1);
        liste.afficher();
        ……
    }
}
```

Une solution d'implantation de pile Lifo générique en C#

```
interface ILifo<T0>
{
    int nbrElt
    {
        get;
    }
    bool estVide();
    void empiler(T0 Elt);
    T0 depiler();
    T0 premier();
}

class Lifo<T0> : ILifo<T0>
{
    public static readonly int max_elt = 100;
    private T0[] t;
    private int sommet, fond;
    public Lifo()
    {
        t = new T0[max_elt];
        fond = -1;
        sommet = fond;
    }
    public int nbrElt
    {
        get { return sommet + 1; }
    }

    public bool estVide()
    {
        return sommet == fond;
    }

    public void empiler(T0 Elt)
    {
        if (sommet < max_elt)
        {
            sommet++;
            t[sommet] = Elt;
        }
        else
            System.Console.WriteLine("empilement impossible : capacité maximale atteinte !");
    }

    public T0 depiler()
    {
        if (!estVide())
        {
            T0 Elt = t[sommet];
            sommet--;
            return Elt;
        }
        else
        {
```

```
            System.Console.WriteLine("dépilement impossible : pile vide !");
            return default(T0);
        }
    }

    public T0 premier()
    {
        if (!estVide())
        {
            return t[sommet];
        }
        else
        {
            System.Console.WriteLine("premier impossible : pile vide !");
            return default(T0);
        }
    }

    public void afficher()
    {
        for (int k = 0; k <= sommet; k++)
            System.Console.Write(t[k] + ", ");
        System.Console.WriteLine(": nbr elmt = " + this.nbrElt);
    }
}
```

Exemple de classe principale testant la classe « Lifo » sur T_0 = **int** :

```
class ProgramListe
{
    static void Main(string[] args)
    {
            Lifo<int> pile = new Lifo<int>();
            pile.afficher();
            pile.depiler();
            System.Console.WriteLine("premier=" + pile.premier());
            for (int i = 0; i < Lifo<int>.max_elt / 10; i++)
              pile.empiler(i);
            pile.afficher();
            System.Console.WriteLine("on dépile : " + pile.depiler());
            System.Console.WriteLine("on dépile : " + pile.depiler());
            pile.afficher();
    }
}
```

Une solution d'implantation de file Fifo générique en C#

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

  interface IFifo<T0>
  {
     int nbrElt
     {
        get;
     }
     bool estVide();
     void ajouter(T0 Elt);
     T0 retirer();
     T0 premier();
  }

  class Fifo<T0> : IFifo<T0>
  {
     public static readonly int max_elt = 100;
     private T0[] t;
     private int tete, fin;

     private void decaleUn()
     {
        if (tete < max_elt)
        {
           tete++;
           for (int k = tete; k >= 0; k--)
              t[k + 1] = t[k];
        }
     }
     public Fifo()
     {
        t = new T0[max_elt];
        tete = -1;
        fin = 0;
     }
     public int nbrElt
     {
        get { return tete + 1; }
     }

     public bool estVide()
     {
        return tete == -1;
     }

     public void ajouter(T0 Elt)
     {
        if (tete < max_elt)
        {
           decaleUn();
           t[fin] = Elt;
        }
```

```
        else
            System.Console.WriteLine("ajout impossible : capacité maximale atteinte !");
    }

    public T0 retirer()
    {
        if (!estVide())
        {
            T0 Elt = t[tete];
            tete--;
            return Elt;
        }
        else
        {
            System.Console.WriteLine("ajout impossible : file vide !");
            return default(T0);
        }
    }

    public T0 premier()
    {
        if (!estVide())
        {
            return t[tete];
        }
        else
        {
            System.Console.WriteLine("premier impossible : file vide !");
            return default(T0);
        }
    }

    public void afficher()
    {
        for (int k = fin; k <= tete; k++)
            System.Console.Write(t[k] + ", ");
        System.Console.WriteLine(": nbr elmt = " + this.nbrElt);
    }
}
class ProgramFifo
{
    static void Main(string[ ] args)
    {
        Fifo<int> file = new Fifo<int>();
        file.afficher();
        file.retirer();
        System.Console.WriteLine("premier = " + file.premier());
        for (int i = 0; i < Fifo<int>.max_elt / 10; i++)
            file.ajouter(i);
        file.afficher();
        System.Console.WriteLine("on retire : " + file.retirer());
        System.Console.WriteLine("on retire : " + file.retirer());
        file.afficher();
    }
}
```

Exemple de classe principale testant la classe « Fifo » sur T_0 = **int** :

```
class ProgramFifo
{
    static void Main(string[] args)
    {
        Fifo<int> file = new Fifo<int>();
        file.afficher();
        file.retirer();
        System.Console.WriteLine("premier = " + file.premier());
        for (int i = 0; i < Fifo<int>.max_elt / 10; i++)
            file.ajouter(i);
        file.afficher();
        System.Console.WriteLine("on retire : " + file.retirer());
        System.Console.WriteLine("on retire : " + file.retirer());
        file.afficher();
    }
}
```

Résultats d'exécution sur la console :

: nbr elmt = 0
ajout impossible : file vide !
premier impossible : file vide !
premier = 0
9, 8, 7, 6, 5, 4, 3, 2, 1, 0, : nbr elmt = 10
on retire : 0
on retire : 1
9, 8, 7, 6, 5, 4, 3, 2, : nbr elmt = 8

Structures d'arbres binaires
et classes génériques

Plan du chapitre:

1. Notions générales sur les arbres

1.1 Vocabulaire *employé sur les arbres :*

- Etiquette Racine, noeud, branche, feuille
- Hauteur, profondeur ou niveau d'un noeud
- Chemin d'un noeud, Noeuds frères, parents, enfants, ancêtres
- Degré d'un noeud
- Hauteur ou profondeur d'un arbre
- Degré d'un arbre
- Taille d'un arbre

1.2 Exemples et implémentation d'arbre

- Arbre de dérivation
- Arbre abstrait
- Arbre lexicographique
- Arbre d'héritage
- Arbre de recherche

2. Arbres binaires

2.1 TAD d'arbre binaire
2.2 Exemples et implémentation d'arbre

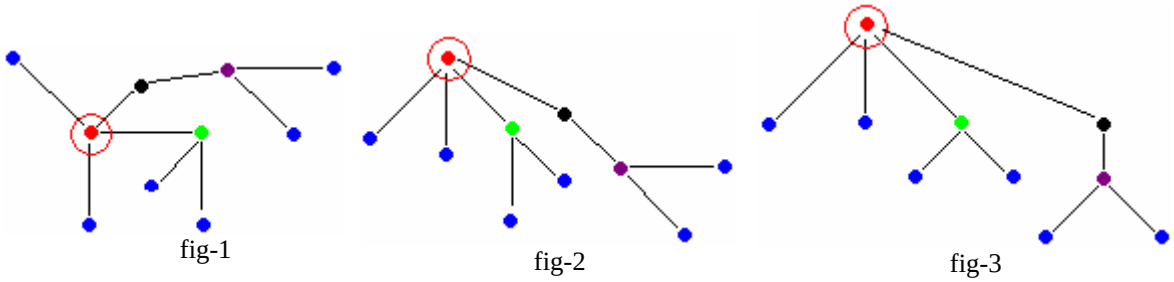
- *tableau statique*
- *variable dynamique*
- *classe*

2.3 Arbres binaires de recherche
2.4 Arbres binaires partiellement ordonnés *(tas)*
2.5 Parcours en largeur et profondeur d'un arbre binaire

- Parcours d'un arbre
- Parcours en largeur
- Parcours préfixé
- Parcours postfixé
- Parcours infixé
- Illustration d'un parcours en profondeur complet
- Exercice

1. Notions générales sur les structures d'arbres

La structure d'arbre est très utilisée en informatique. Sur le fond on peut considérer un arbre comme une généralisation d'une liste car les listes peuvent être représentées par des arbres. La complexité des algorithmes d'insertion de suppression ou de recherche est généralement plus faible que dans le cas des listes (cas particulier des arbres équilibrés). Les mathématiciens voient les arbres eux-même comme des cas particuliers de graphes non orientés connexes et acycliques, donc contenant des sommets et des arcs :



Ci dessus 3 représentations graphiques de la même structure d'arbre : dans la figure fig-1 tous les sommets ont une disposition équivalente, dans la figure fig-2 et dans la figure fig-3 le sommet "**cerclé**" se distingue des autres.

Lorsqu'un sommet est distingué par rapport aux autres, on le dénomme **racine** et la même structure d'arbre s'appelle une **arborescence**, par abus de langage dans tout le reste du document nous utliserons le vocable **arbre** pour une **arborescence**.

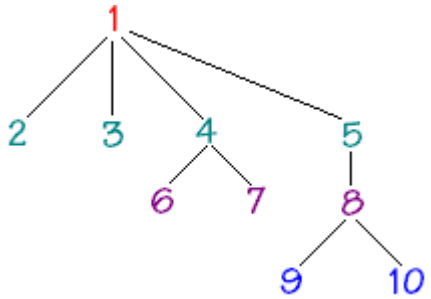
Enfin certains arbres particuliers nommés arbres binaires sont les plus utilisés en informatique et les plus simples à étudier. En outre il est toujours possible de "**binariser**" un arbre non binaire, ce qui nous permettra dans ce chapitre de n'étudier que les structures d'arbres binaires.

1.1 Vocabulaire employé sur les arbres

Etiquette

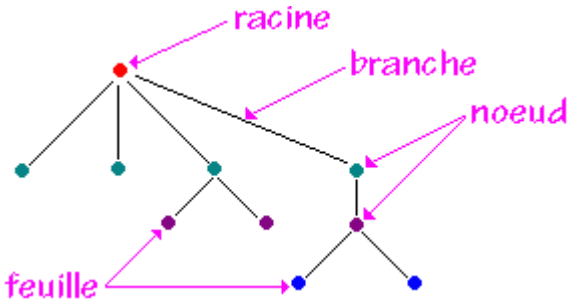
Un arbre dont tous les noeuds sont nommés est dit **étiqueté**. L'étiquette (ou nom du sommet) représente la "valeur" du noeud ou bien l'information associée au noeud.
Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :



Nous rappellons la terminologie de base sur les arbres:

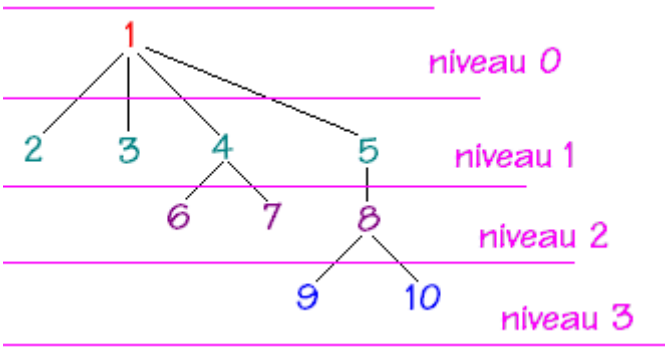
Racine , noeud, branche , feuille



Hauteur d'un noeud

Nous conviendrons de définir la **hauteur** (ou **profondeur** ou **niveau d'un noeud**) d'un noeud X comme égale **au nombre de noeuds à partir de la racine** pour aller jusqu'au noeud X.

En reprenant l'arbre précédant et en notant **h** la fonction hauteur d'un noeud :



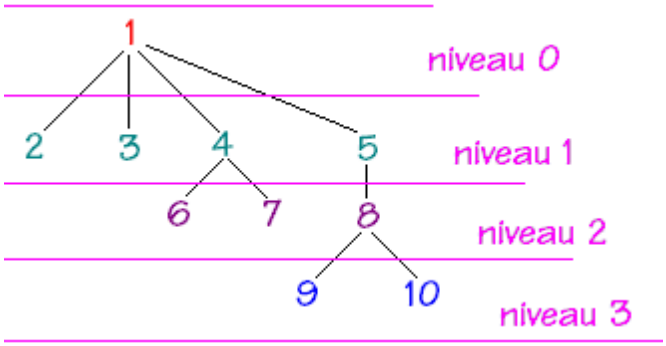
Pour atteindre le noeud étiqueté 9 , il faut parcourir le lien 1--5, puis 5--8, puis enfin 8--9 soient 4 noeuds donc 9 est de profondeur ou de hauteur égale à 4, soit **h(9) = 4**.
Pour atteindre le noeud étiqueté 7 , il faut parcourir le lien 1--4, et enfin 4--7, donc 7 est de profondeur ou de hauteur égale à 3, soit **h(7) = 3**.

Par définition la hauteur de la racine est égal à 1.
h(racine) =1 *(pour tout arbre non vide)*

(Certains auteurs adoptent une autre convention pour calculer la hauteur d'un noeud: la racine a pour hauteur 0 et donc n'est pa comptée dans le nombre de noeuds, ce qui donne une hauteur inférieure d'une unité à notre définition).

Chemin d'un noeud

On appelle chemin du noeud X la **suite des noeuds** par lesquels il faut passer pour aller de la racine vers le noeud X.

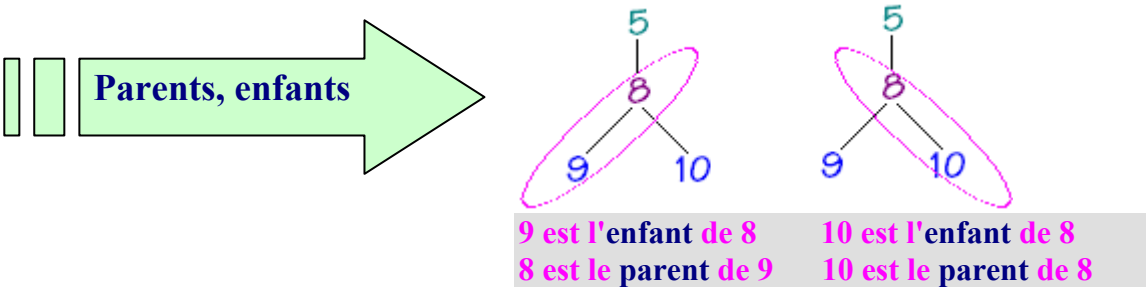


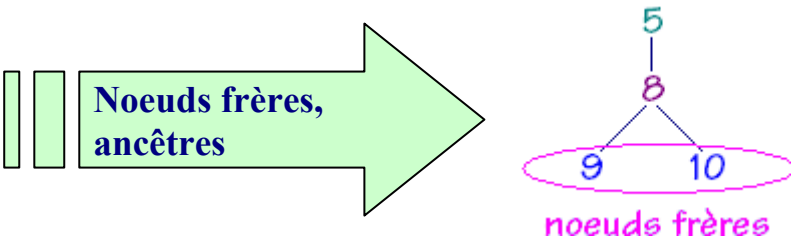
Chemin du noeud 10 = (1,5,8,10)
Chemin du noeud 9 = (1,5,8,9)
.....
Chemin du noeud 7 = (1,4,7)
Chemin du noeud 5 = (1,5)
Chemin du noeud 1 = (1)

Remarquons que la hauteur h d'un noeud X est égale au nombre de noeuds dans le chemin :

h(X) = NbrNoeud(Chemin(X)).

Le vocabulaire de lien entre noeuds de niveau différents et reliés entres eux est emprunté à la généalogie :





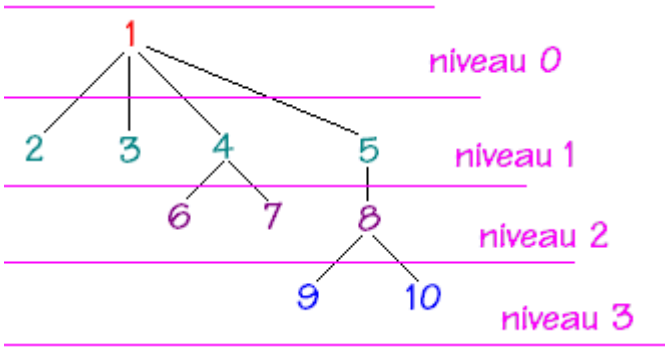
- **9 et 10 sont des frères**
- **5 est le parent de 8 et l'ancêtre de 9 et 10.**

On parle aussi d'ascendant, de descendant ou de fils pour évoquer des relations entres les noeuds d'un même arbre reliés entre eux.

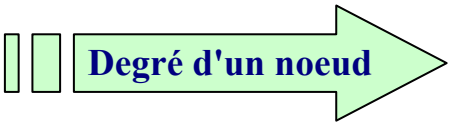
Nous pouvons définir récursivement la hauteur h d'un noeud X à partir de celle de son parent :

h (racine) = 1;
h (X) = 1+ h (parent (X))

Reprenons l'arbre précédent en exemple :

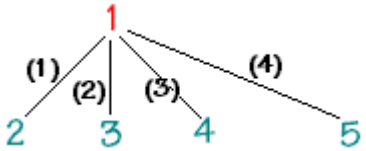


Calculons récursivement la hauteur du noeud 9, notée h(9) :
h(9) = 1+h(8)
h(8) = 1+h(5)
h(5) = 1+h(1)
h(1) = 1 = h(5)=2 = h(8)=3 = h(9)=4

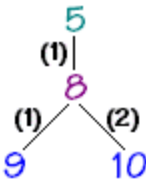


Par définition le **degré** d'un noeud est égal au **nombre de ses descendants** (enfants).

Soient les deux exemples ci-dessous extraits de l'arbre précédent :

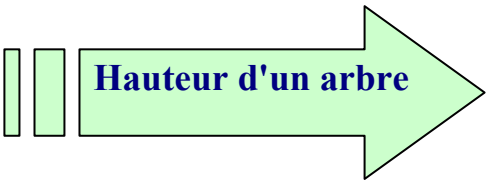


Le noeud 1 est de degré 4, car il a 4 enfants



Le noeud 5 n'ayant qu'un enfant son degré est 1.
Le noeud 8 est de degré 2 car il a 2 enfants.

Remarquons
que lorsqu'un arbre a **tous ses noeuds de degré 1**, on le nomme **arbre dégénéré** et que c'est en fait une **liste**.



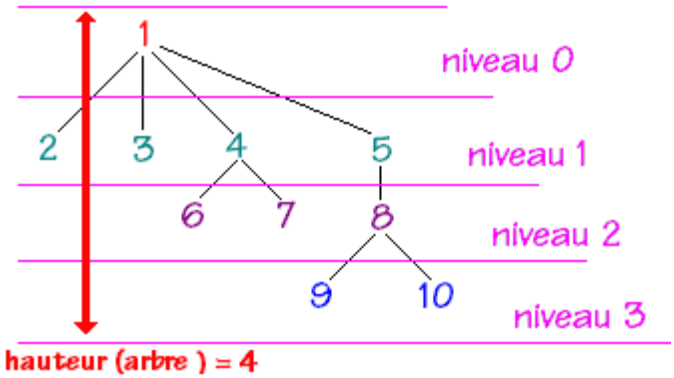
Par définition c'est le **nombre de noeuds du chemin le plus long** dans l'arbre.; on dit aussi profondeur de l'arbre.

La hauteur **h** d'un arbre correspond donc au nombre maximum de niveaux :

Hauteur d'un arbre

h (Arbre) = max { h (X) / ∀ X, X noeud de Arbre }
si Arbre = ∅ alors h(Arbre) = 0

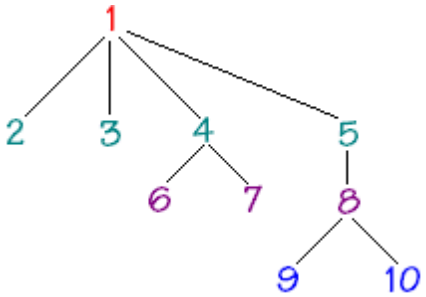
La hauteur de l'arbre ci-dessous :



Degré d'un arbre

Le degré d'un arbre est égal au **plus grand des degrés de ses nœuds :**
d°(Arbre) = max { d°(X) / ∀ X, X noeud de Arbre }

Soit à répertorier dans l'arbre ci-dessous le degré de chacun des noeuds :



d°(1) = 4 d°(2) = 0
d°(3) = 0 d°(4) = 2
d°(5) = 1 d°(6) = 0
d°(7) = 0 d°(8) = 2
d°(9) = 0 d°(10) = 0

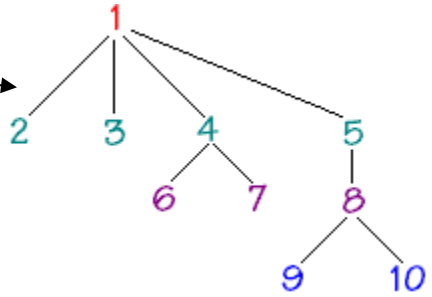
La valeur maximale est 4 , donc cet arbre est de degré 4.

Taille d'un arbre

On appelle **taille** d'un arbre le nombre total de noeuds de cet arbre :

taille(< r , fg , fd >) = 1 + taille(fg) + taille(fd)

L'arbre ci-contre a pour **taille** 10 (car il a 10 noeuds)



1.2 Exemples et implémentation d'arbre

Les structures de données arborescentes permettent de représenter de nombreux problèmes, nous proposons ci-après quelques exemples d'utilisations d'arbres dans des contextes différents.

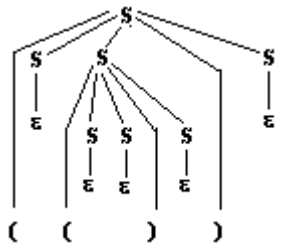
Arbre de dérivation d'un mot dans une grammaire

Exemple - 1
arbre d'analyse

```
Soit la grammaire G2 : VN = {S}
VT = {( , )}
Axiome : S
Règles
   1 : S → (SS)S
   2 : S → ε
```

Le langage $L(G_2)$ se dénomme langage des parenthèses bien formées.

Soit le mot (( )) de G_2 , voici un arbre de dérivation de (( )) dans G_2 :



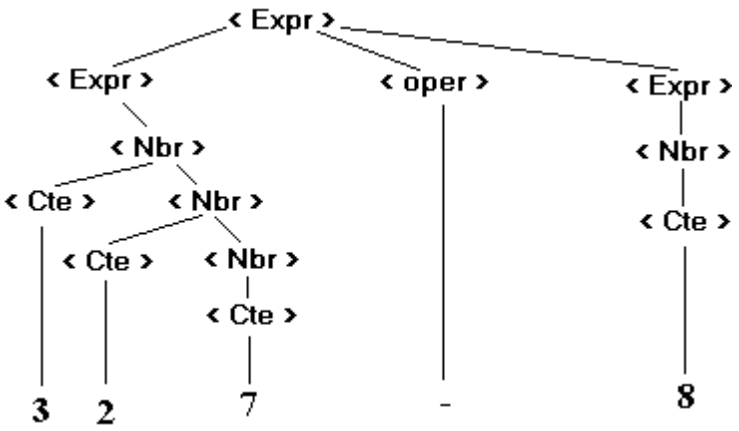
Exemple - 2
arbre abstrait

Soit la grammaire Gexp :

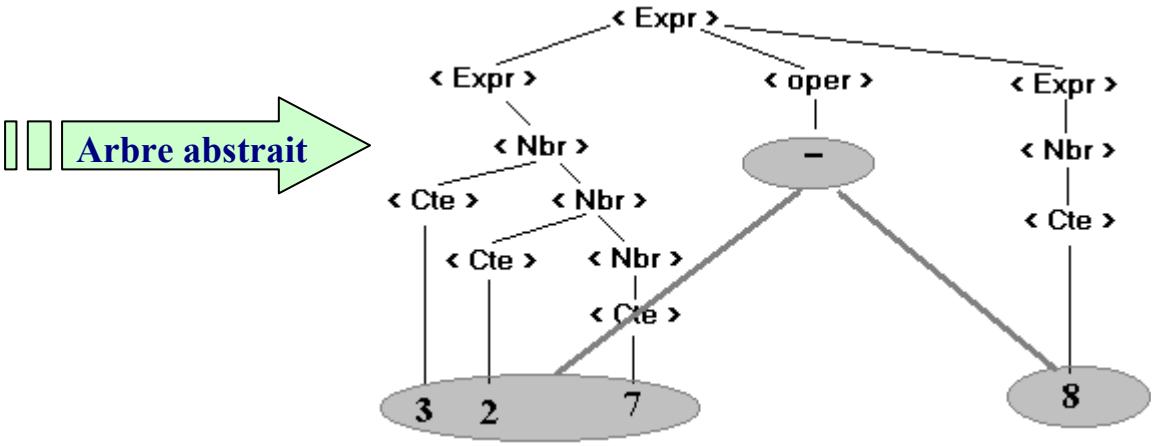
```
Gexp = (VN,VT,Axiome,Règles)
  VT = { 0, ..., 9, +,-, /, *, ), (}
  VN = { ⟨ Expr ⟩, ⟨ Nbr ⟩, ⟨ Cte ⟩, ⟨ Oper ⟩}
  Axiome : ⟨ Expr ⟩
  Règles :
  1 :⟨ Expr ⟩→⟨ Nbr ⟩ | (⟨Expr ⟩)| ⟨ Expr ⟩⟨ Oper ⟩⟨ Expr ⟩
  2 :⟨ Nbr ⟩→⟨ Cte ⟩ | ⟨ Cte ⟩⟨ Nbr ⟩
  3 :⟨ Cte ⟩→0 | 1 |...| 9
  4 :⟨ Oper ⟩→+ | - | * | /
```

soit : **327 - 8** un mot de $L(G_{exp})$

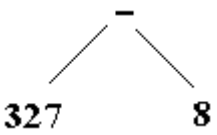
Soit son arbre de dérivation dans G_{exp} :



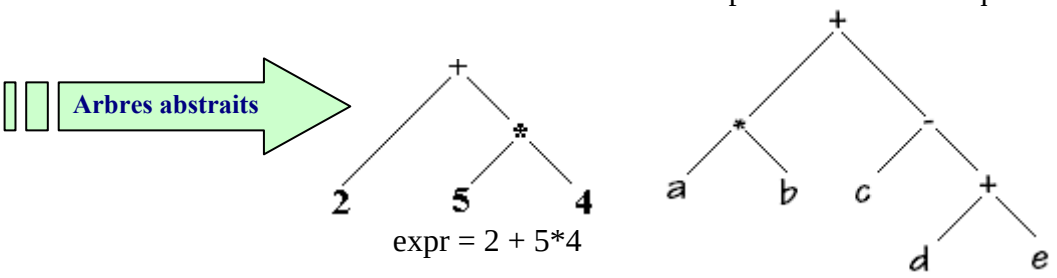
L'arbre obtenu ci-dessous en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 " :



On note ainsi cet arbre abstrait :

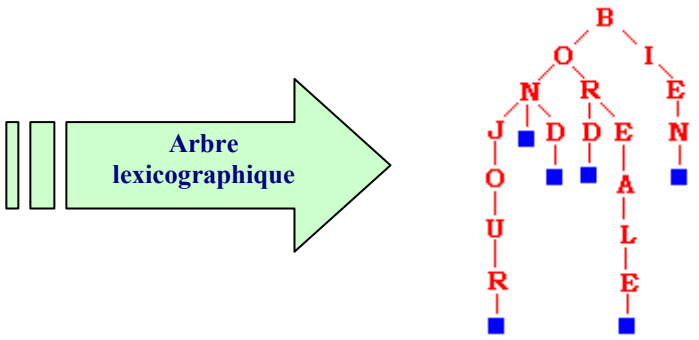


Voici d'autres abres abstraits d'expressions arithmétiques :

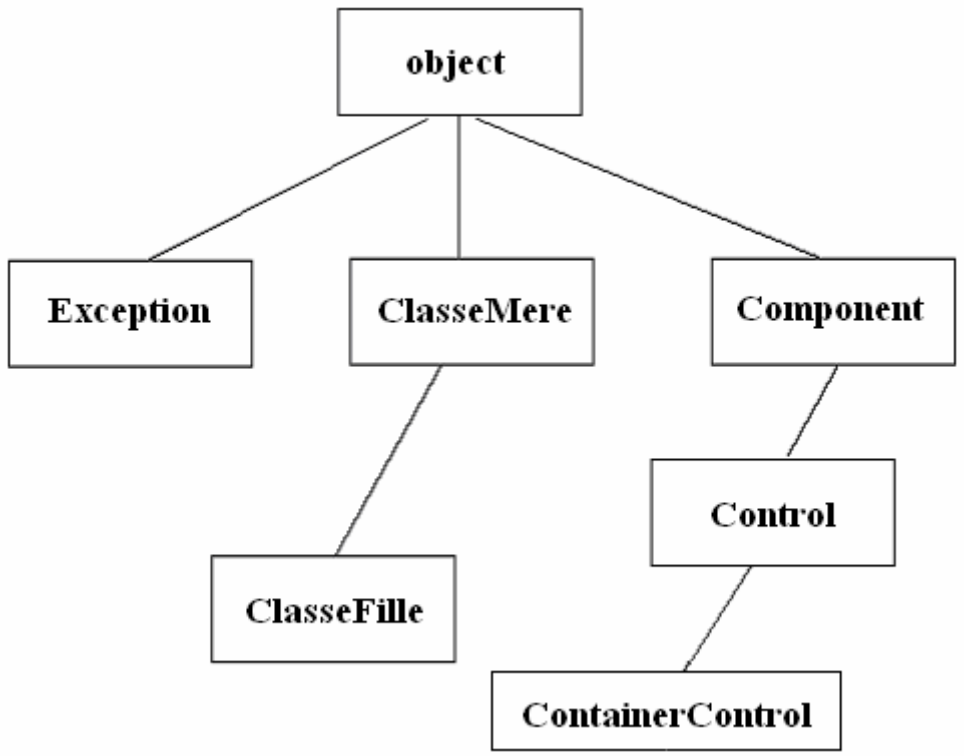


Arbre lexicographique

Rangement de mots par ordre lexical (alphabétique)
Soient les mots BON, BONJOUR, BORD, BOND, BOREALE, BIEN, il est possible de les ranger ainsi dans une structure d'arbre :



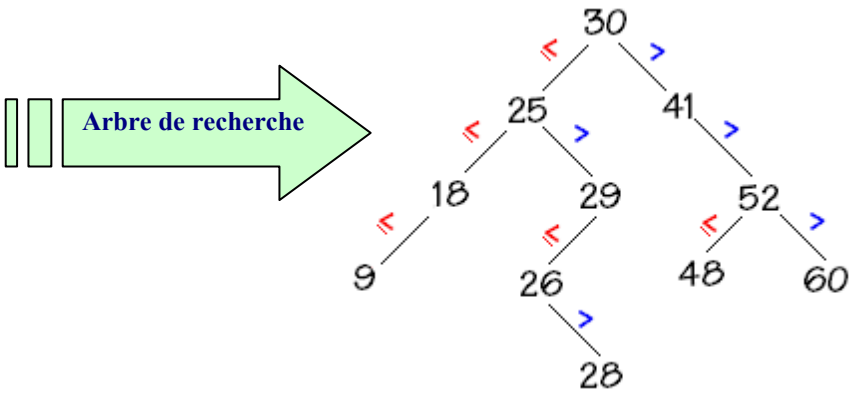
Arbre d'héritage en C#



Arbre de recherche

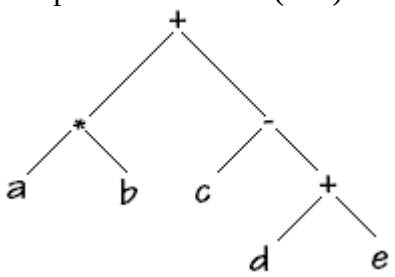
Voici à titre d'exemple que nous étudierons plus loin en détail, un arbre dont les noeuds sont de degré 2 au plus et qui est tel que pour chaque noeud la valeur de son enfant de gauche lui est inférieure ou égale, la valeur de son enfant de droite lui est strictement supérieure.

Ci-après un tel arbre ayant comme racine 30 et stockant des entiers selon cette répartition :

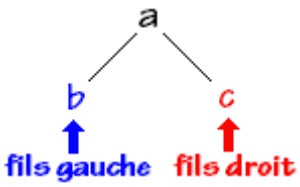


2 Les arbres binaires

Un arbre **binaire** est un arbre de degré 2 (dont les noeuds sont de degré 2 au plus).
L'arbre abstrait de l'expression a*b + c-(d+e) est un arbre binaire :



Vocabulaire :
Les descendants (enfants) d'un noeud sont lus de gauche à droite et sont appelés respectivement **fils gauche** (descendant gauche) et **fils droit** (descendant droit) de ce noeud.



Les arbres binaires sont utilisés dans de très nombreuses activités informatiques et comme nous l'avons déjà signalé il est toujours possible de reprrésenter un arbre général (de degré 2) par un arbre binaire en opérant une "binarisation".

Nous allons donc étudier dans la suite, le comportement de cette structure de donnée récursive.

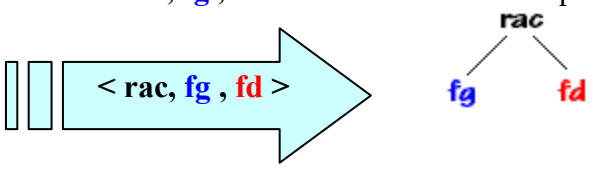
2.1 TAD d'arbre binaire

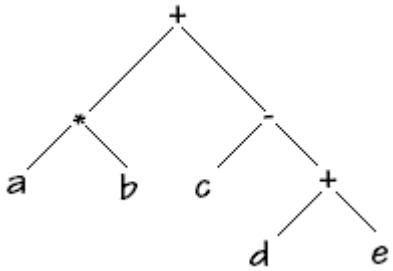
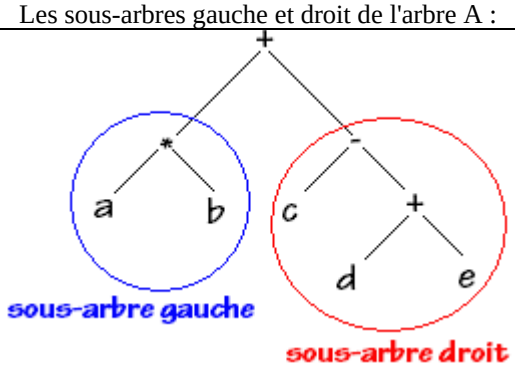
Afin d'assurer une cohérence avec les autres structures de données déjà vues (**liste, pile, file**) nous proposons de décrire une abstraction d'un arbre binaire avec un TAD. Soit la signature du TAD d'arbre binaire :

```
TAD ArbreBin
utilise : T0, Noeud, Booleens
opérations :
   ∅   : → ArbreBin
   Racine : ArbreBin → Noeud
   filsG : ArbreBin → ArbreBin
   filsD : ArbreBin → ArbreBin
   Constr : Noeud x ArbreBin x ArbreBin → ArbreBin
   Est_Vide  : ArbreBin → Booleens
   Info : Noeud → T0

préconditions :

   Racine(Arb) def_ssi Arb ≠ ∅
   filsG(Arb)  def_ssi Arb ≠ ∅
   filsD(Arb)  def_ssi Arb ≠ ∅
axiomes :

    ∀rac ∈ Noeud ,  ∀fg ∈ ArbreBin , ∀fd ∈ ArbreBin
    Racine(Constr(rac,fg,fd)) = rac
    filsG(Constr(rac,fg,fd))  = fg
    filsD(Constr(rac,fg,fd))  = fd
    Info(rac) ∈T0

FinTAD- ArbreBin
```

- T_0 est le type des données rangées dans l'arbre.
- L'opérateur **filsG( ) renvoie le sous-arbre gauche** de l'arbre binaire, l'opérateur **filsD( ) renvoie le sous-arbre droit** de l'arbre binaire, l'opérateur Info( ) permet de stocker des informations de type T_0 dans chaque noeud de l'arbre binaire.

Nous noterons **< rac, fg , fd >** avec conventions implicites un arbre binaire dessiné ci-dessous :



Exemple, soit l'arbre binaire A :	Les sous-arbres gauche et droit de l'arbre A :
A = 	 filsG(A) = < * , a , b > filsD(A) = < - , c , < + , d , e > >

2.2 Exemples et implémentation d'arbre binaire étiqueté

Nous proposons de représenter un **arbre binaire étiqueté** selon deux spécifications différentes classiques :

1°) Une implantation fondée sur une structure de tableau en allocation de mémoire statique, nécessitant de connaître au préalable le nombre maximal de noeuds de l'arbre (ou encore sa taille).

2°) Une implantation fondée sur une structure d'allocation de mémoire dynamique implémentée soit par des pointeurs (variables dynamiques) soit par des références (objets) .

Implantation dans un tableau statique

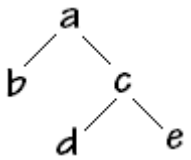
Spécification concrète
Un noeud est une structure statique contenant 3 éléments :

- l'information du noeud
- le fils gauche
- le fils droit

Pour un arbre binaire de taille = n, **chaque noeud de l'arbre binaire est stocké dans une cellule d'un tableau** de dimension 1 à n cellules. Donc chaque noeud est repéré dans le tableau par un indice (celui de la cellule le contenant).
Le champ fils gauche du noeud sera l'indice de la cellule contenant le descendant gauche, et le champ fils droit vaudra l'indice de la cellule contenant le descendant droit.

Exemple

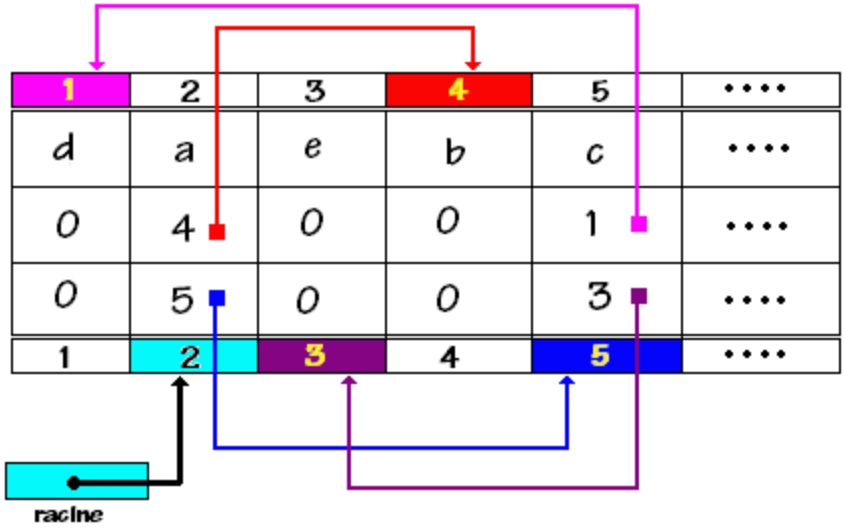
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, la racine <a, vers b, vers c >est contenue dans la cellule d'indice 2 du tableau, les autres noeuds sont supposés être rangés dans les cellules 1, 3,4,5 :

Nous avons :

racine = table[2]
table[1] = < d , 0 , 0 >
table[2] = < a , **4** , **5** >
table[3] = < e , 0 , 0 >
table[4] = < b , 0 , 0 >
table[5] = < c , **1** , **3** >



Explications :

table[2] = < a , **4** , **5** >signifie que le fils gauche de ce noeud est dans table[**4**] et son fils droit dans table[**5**]
table[5] = < c , **1** , **3** >signifie que le fils gauche de ce noeud est dans table[**1**] et son fils droit dans table[**3**]
table[1] = < d , 0 , 0 > signifie que ce noeud est une feuille
...etc

Implantation dynamique avec une classe

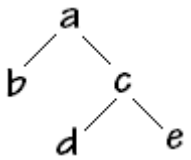
Spécification concrète

Le noeud est un objet contenant 3 éléments dont 2 sont eux-mêmes des objets de noeud :

- l'information du noeud
- une référence vers le fils gauche
- une référence vers le fils droit

Exemple

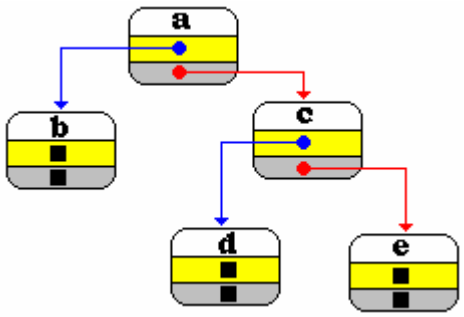
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, le premier objet appelé racine de l'arbre est **< a, ref vers b, ref vers c >**

Nous avons :

racine → < a, ref vers b, ref vers c >
ref vers b → < b, null, null >
ref vers c → < a, ref vers d, ref vers e >
ref vers d → < d, null, null >
ref vers e → < e, null, null >



Spécification d'implantation en C#

Nous livrons ci-dessous une écriture de la signature et l'implémentation minimale d'une classe d'arbre binaire nommée ArbreBin en C# dans laquelle les informations portées par un nœud sont de type T0 :

```
interface IArbreBin<T0>
{
   T0 Info {    get;      set;       }
}
class ArbreBin<T0> : IArbreBin<T0>
{
   private T0 InfoLoc;
```

```
    private ArbreBin<T0> fg;
    private ArbreBin<T0> fd;

    public ArbreBin(T0 s) : this ( )       {
       InfoLoc = s;
    }
    public ArbreBin ( )
    {
       InfoLoc = default(T0);
       fg = default(ArbreBin<T0>);
       fd = default(ArbreBin<T0>);
    }
    public T0 Info
    {
       get   {  return InfoLoc;   }
       set    {  InfoLoc = value; }
    }

    public ArbreBin<T0> filsG
    {
       get   {  return this.fg;  }
       set   {  fg = new ArbreBin<T0>(value.Info); }
    }
    public ArbreBin<T0> filsD
    {
       get  { return this.fd;  }
       set  {  fd = new ArbreBin<T0>(value.Info); }
    }
  }
```

2.3 Arbres binaires de recherche

- Nous avons étudié prédédement des algorithmes de recherche en table, en particulier la recherche dichotomique dans une table triée dont la recherche s'effectue en **O(log(n))** comparaisons.
- Toutefois lorsque le nombre des éléments varie (ajout ou suppression) ces ajouts ou suppressions peuvent nécessiter des temps en **O(n)**.
- En utilisant une liste chaînée qui approche bien la structure dynamique (plus gourmande en mémoire qu'un tableau) on aura en moyenne des temps de suppression ou de recherche au pire de l'ordre de **O(n)**. L'ajout en fin de liste ou en début de liste demandant un temps constant noté **O(1)**.

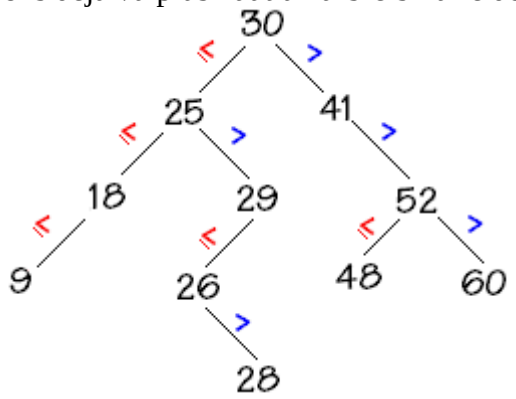
Les arbres binaires de recherche sont un bon compromis pour un temps **équilibré entre ajout, suppression et recherch**e.

Un arbre binaire de recherche satisfait aux critères suivants :

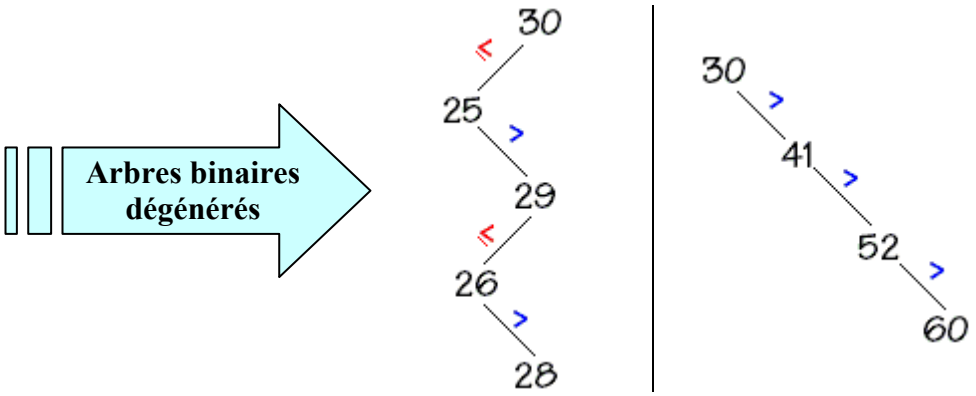
- L'ensemble des étiquettes est **totalement ordonné**.
- Une étiquette est dénommée **clef**.
- Les **clefs** de tous les noeuds du sous-arbre **gauche** d'un noeud X, sont **inférieures ou égales** à la clef de X.

- Les **clefs** de tous les noeuds du sous-arbre **droit** d'un noeud X, sont **supérieures** à la clef de X.

Nous avons déjà vu plus haut un arbre binaire de recherche :



Prenons par exemple le noeud (25) son sous-arbre droit est bien composé de noeuds dont les clefs sont supérieures à 25 : (29,26,28). Le sous-arbre gauche du noeud (25) est bien composé de noeuds dont les clefs sont inférieures à 25 : (18,9).
On appelle arbre binaire dégénéré un arbre binaire dont le degré = 1, ci-dessous 2 arbres binaires de recherche dégénérés :



Nous remarquons dans les deux cas que nous avons affaire à une liste chaînée donc le nombre d'opération pour la suppression ou la recherche est au pire de l'ordre de O(n).

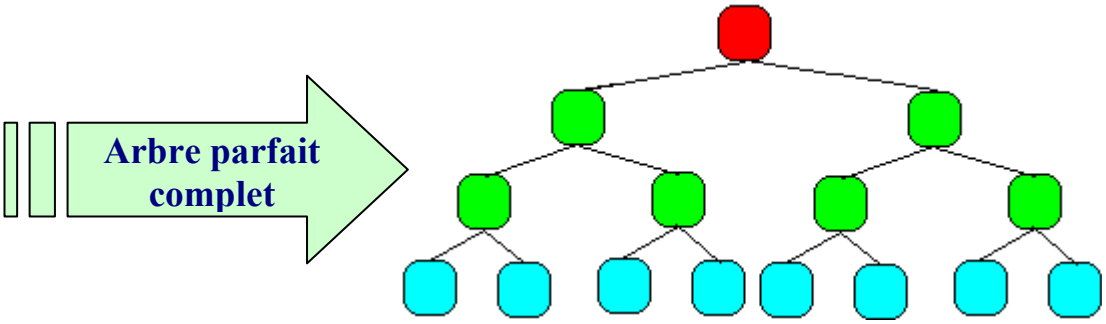
Il faudra donc utiliser une catégorie spéciale d'arbres binaires qui restent équilibrés (leurs feuilles sont sur 2 niveaux au plus) pour assurer une recherche au pire en O(log(n)).

2.4 Arbres binaires partiellement ordonnés (tas)

Nous avons déjà évoqué la notion d'arbre parfait lors de l'étude du tri par tas, nous récapitulons ici les éléments essentiels le lecteur

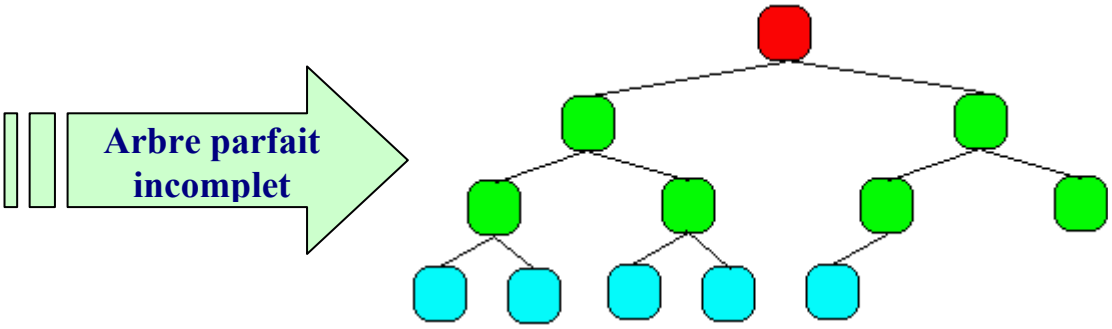
c'est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux

= feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet et les feuilles du dernier niveau **doivent être regroupées à partir de la gauche** de l'arbre.



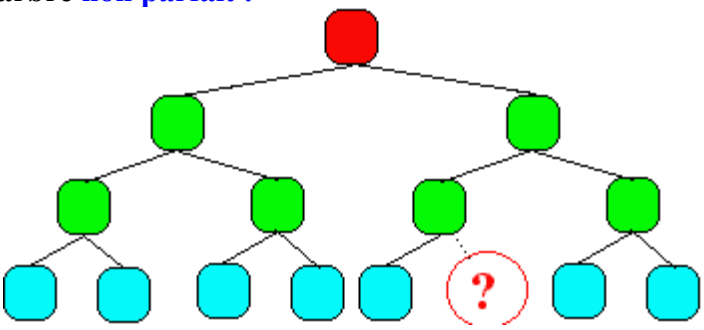
parfait complet : *le dernier niveau est complet car il contient tous les enfants*

un arbre **parfait** *peut être incomplet lorsque le dernier niveau de l'arbre est incomplet (dans le cas où manquent des feuilles à la droite du dernier niveau, les feuilles sont regroupées à gauche)*



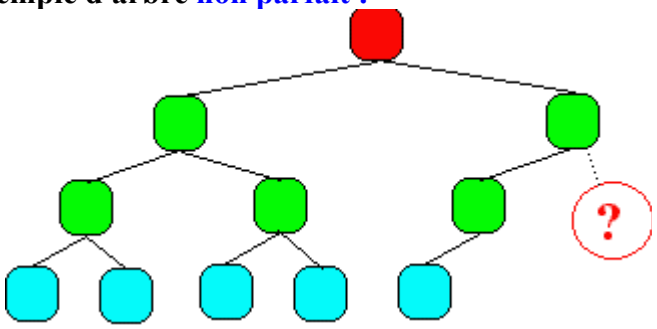
parfait incomplet: *le dernier niveau est incomplet car il manque 3 enfants à la droite*

Exemple d'arbre non parfait :



(**non parfait :** *les feuilles ne sont pas regroupées à gauche)*

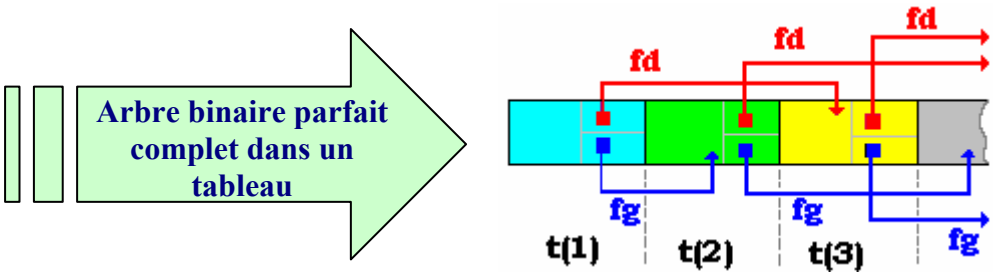
Autre exemple d'arbre non parfait :



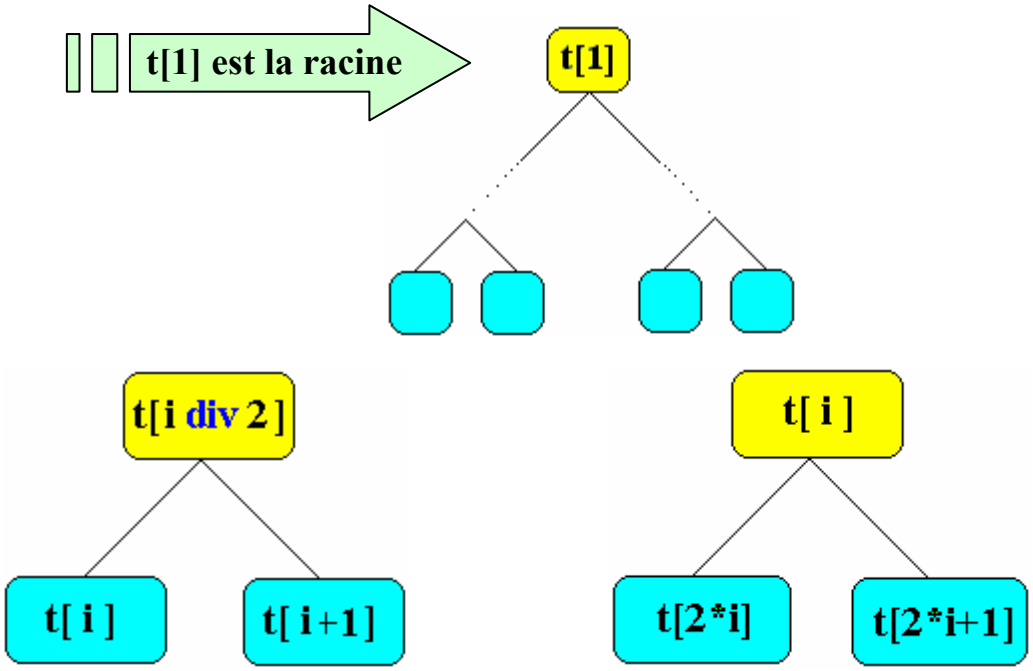
(**non parfait :** *les feuilles sont bien regroupées à gauche, mais il manque 1 enfant à l'avant dernier niveau*)

Un arbre binaire parfait se représente classiquement dans un tableau :

Les noeuds de l'arbre sont dans les cellules du tableau, il n'y a pas d'autre information dans une cellule du tableau, l'accès à la topologie arborescente est simulée à travers un calcul d'indice permettant de parcourir les cellules du tableau selon un certain 'ordre' de numérotation correspondant en fait à un **parcours hiérarchique** de l'arbre. En effet ce sont les numéros de ce parcours qui servent d'indice aux cellules du tableau nommé **t** ci-dessous :



Si **t** est ce tableau, nous avons les règles suivantes :

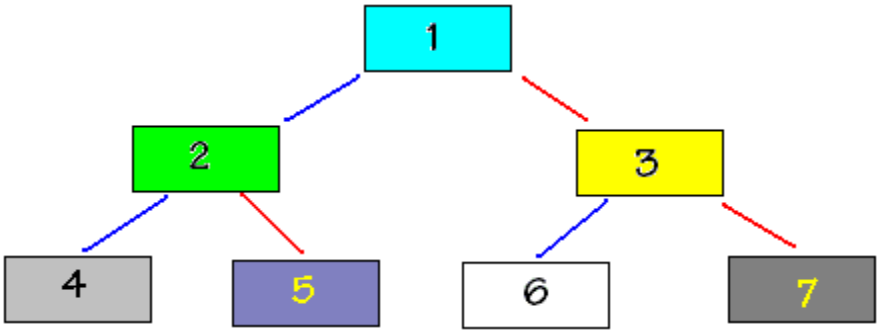


t[i **div** 2] est le père de t[i] pour i > 1

t[2 * i] et **t**[2 * i + 1] sont les deux fils, s'ils existent, de **t**[i]

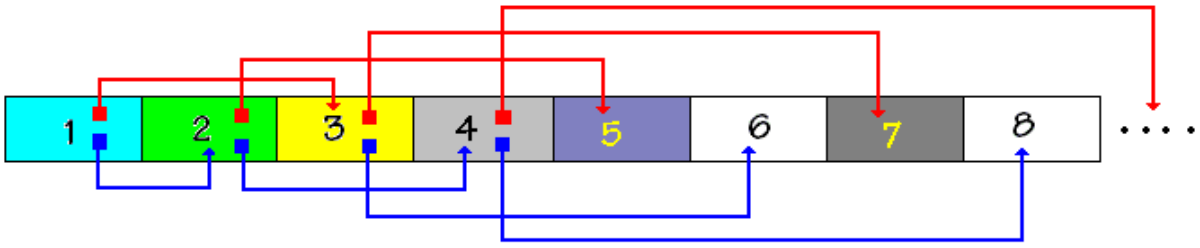
si **p** est le nombre de noeuds de l'arbre et si 2 * i = p, t[i] n'a qu'un fils, t[p].
si i est supérieur à p **div** 2, t[i] est une feuille.

Exemple de rangement d'un tel arbre dans un tableau
(on a figuré l'indice de numérotation hiérarchique de chaque noeud dans le rectangle associé au noeud)



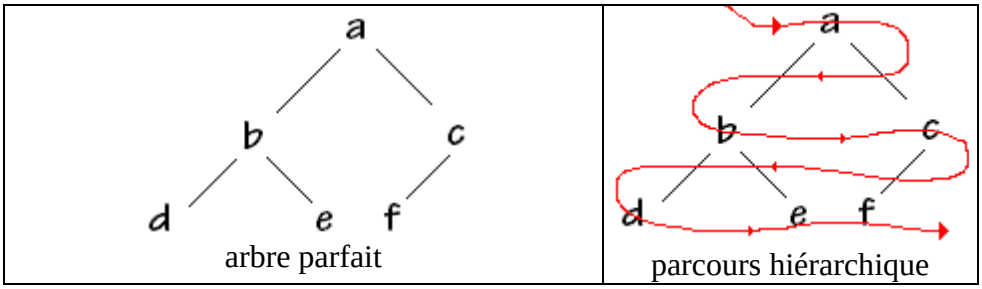
Cet arbre sera stocké dans un tableau en disposant séquentiellement et de façon contigüe les noeuds selon la numérotation hiérarchique (l'index de la cellule = le numéro hiérarchique du noeud).

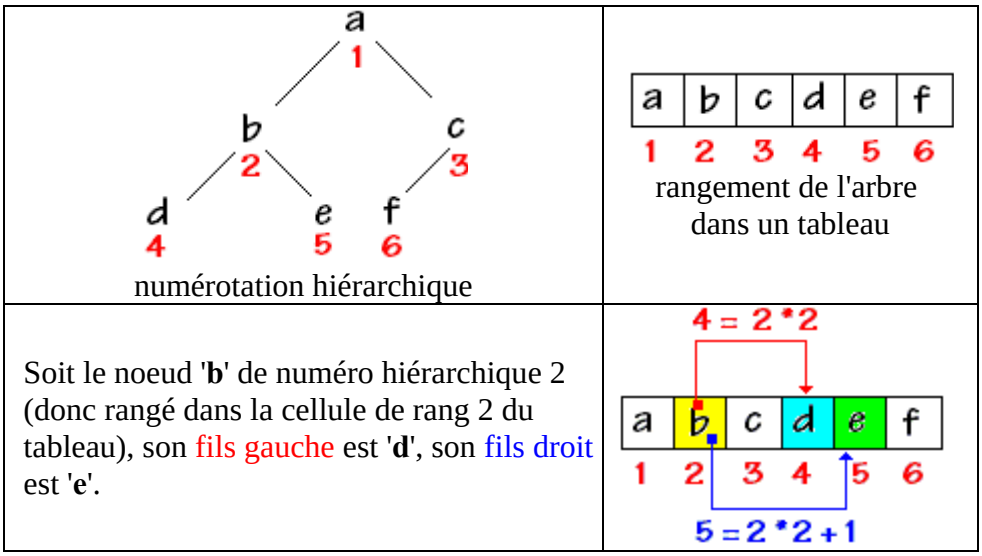
Dans cette disposition le passage d'un noeud de numéro **k** (indice dans le tableau) vers son fils gauche s'effectue par calcul d'indice, le fils gauche se trouvera dans la cellule d'index **2*k** du tableau, son fils droit se trouvant dans la cellule d'index **2*k + 1** du tableau. Ci-dessous l'arbre précédent est stocké dans un tableau : le noeud d'indice hiérarchique 1 (la racine) dans la cellule d'index 1, le noeud d'indice hiérarchique 2 dans la cellule d'index 2, etc...
Le nombre qui figure dans la cellule (nombre qui vaut l'index de la cellule = le numéro hiérarchique du noeud) n'est mis là qu'à titre pédagogique afin de bien comprendre le mécanisme.



On voit par exemple, que par calcul on a bien le fils gauche du noeud d'indice 2 est dans la cellule d'index 2*2 = 4 et son fils droit se trouve dans la cellule d'index 2*2+1 = 5 ...

Exemple d'un arbre parfait étiqueté avec des caractères :





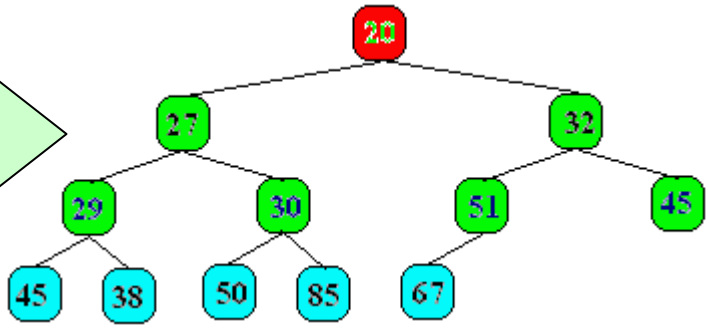
Arbre partiellement ordonné

C'est un arbre étiqueté dont les valeurs des noeuds appartiennent à un ensemble muni d'une **relation d'ordre total** (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses **fils ont une valeur supérieure ou égale à celle de leur père**.

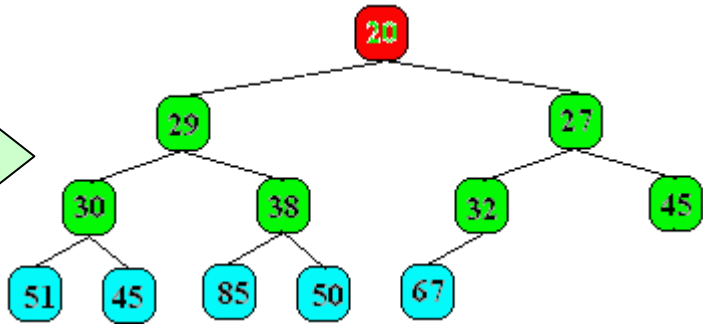
*Exemple de **deux** arbres partiellement ordonnés*

sur l'ensemble {20,27,29,30,32,38,45,45,50,51,67,85} d'entiers naturels :

Arbre partiellement ordonné n°1

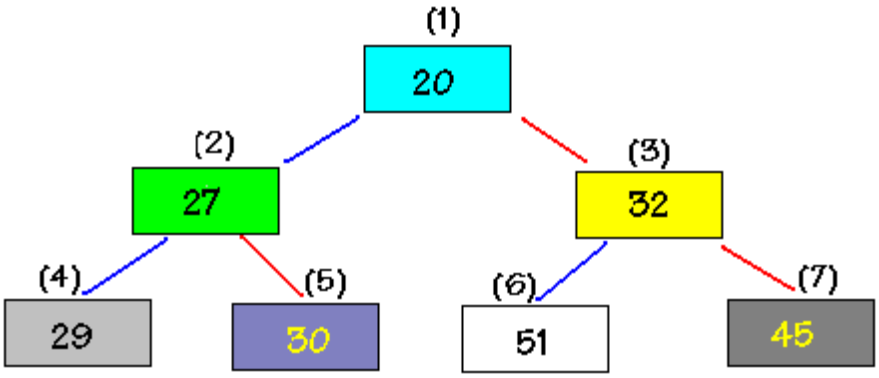


Arbre partiellement ordonné n°2

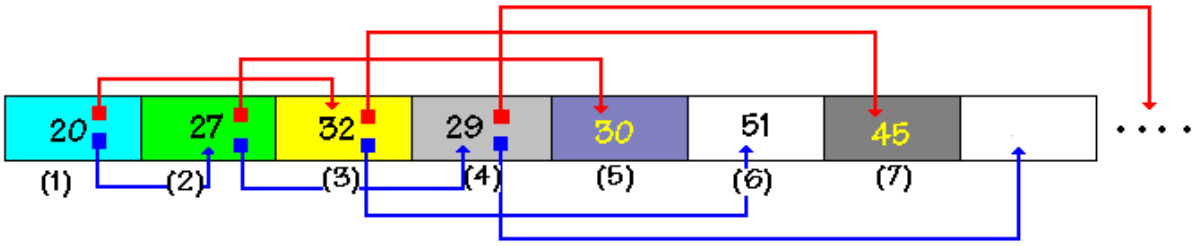


Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses

descendants c'est le minimum. Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre.
En reprenant l'exemple précédent sur 3 niveaux : *(entre parenthèses le numéro hiérarchique du noeud)*



Voici réellement ce qui est stocké dans le tableau : *(entre parenthèses l'index de la cellule contenant le noeud)*



Le tas

On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

L'intérêt d'utiliser un arbre parfait complet ou incomplet réside dans le fait que le tableau est toujours **compacté**, les cellules vides s'il y en a se situent à la fin du tableau.
Le fait d'être partiellement ordonné sur les valeurs permet d'avoir immédiatement un **extremum** à la racine.

2.5 Parcours d'un arbre binaire

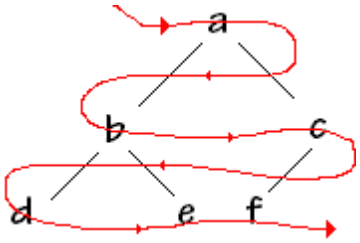
Objectif : les arbres sont des structures de données. Les informations sont contenues dans les noeuds de l'arbre, afin de construire des algorithmes effectuant des opérations sur ces informations (ajout, suppression, modification,...) il nous faut pouvoir examiner tous les noeuds d'un arbre. Examinons les différents moyens de parcourir ou de traverser chaque noeud de l'arbre et d'appliquer un traitement à la donnée rattachée à chaque noeud.

Parcours d'un arbre

L'opération qui consiste à **retrouver** systématiquement tous les noeuds d'un arbre et d'y appliquer un **même traitement** se dénomme **parcours** de l'arbre.

Parcours en largeur ou hiérarchique

Un algorithme classique consiste à **explorer** chaque noeud d'un niveau donné de **gauche à droite**, puis de passer au niveau suivant. On dénomme cette stratégie le parcours en largeur de l'arbre.



Parcours en profondeur

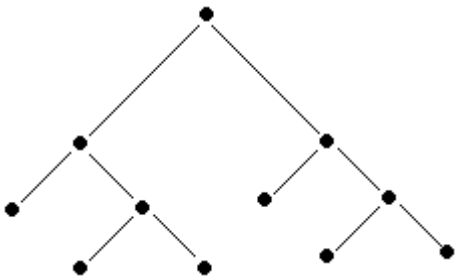
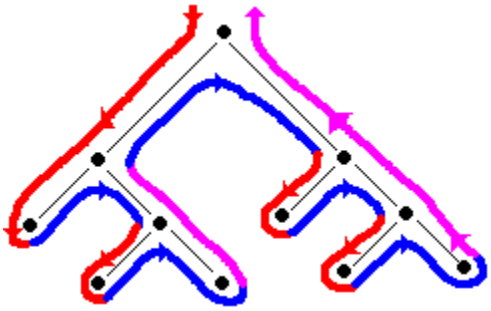
La stratégie consiste à **descendre** le plus profondément soit **jusqu'aux feuilles** d'un noeud de l'arbre, puis lorsque toutes les feuilles du noeud ont été visitées, l'algorithme "**remonte**" au noeud plus haut dont les feuilles n'ont pas encore été visitées.

Notons que ce parcours peut s'effectuer systématiquement en commençant par le fils gauche, puis en examinant le fils droit ou bien l'inverse.

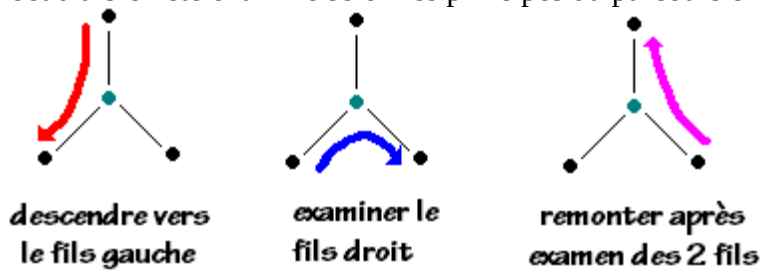
Parcours en profondeur par la gauche

Traditionnellement c'est l'exploration **fils gauche, puis ensuite fils droit** qui est retenue on dit alors que l'on traverse l'arbre en "**profondeur par la gauche**".

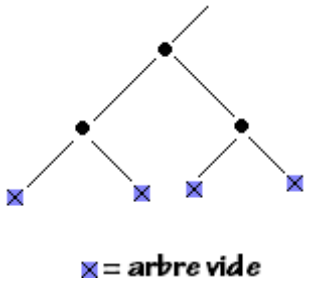
Schémas montrant le principe du parcours exhaustif en "**profondeur par la gauche**" :

Soit l'arbre binaire suivant:	Appliquons la méthode de parcours proposée :
	

Chaque noeud a bien été examiné selon les principes du parcours en profondeur :



En fait pour ne pas surcharger les schémas arborescents, nous omettons de dessiner à la fin de chaque noeud de type feuille les deux **noeuds enfants vides** qui permettent de reconnaître que le parent est une feuille :



Lorsque la compréhension nécessitera leur dessin nous conseillons au lecteur de faire figurer explicitement dans son schéma arborescent les noeuds vides au bout de chaque feuille.

Nous proposons maintenant, de donner une description en langage algorithmique LDFA du parcours en profondeur d'un arbre binaire sous forme récursive.

Algorithme général récursif de parcours en profondeur par la gauche

```
parcourir ( Arbre )
si Arbre ≠ ∅  alors
   Traiter-1 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsG ) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```

Les différents traitements Traiter-1 ,Traiter-2 et Traiter-3 consistent à traiter l'information située dans le noeud actuellement traversé soit lorsque l'on descend vers le fils gauche (Traiter-1), soit en allant examiner le fils droit (Traiter-2), soit lors de la remonté après examen des 2 fils (Traiter-3).

En fait on n'utilise en pratique que trois variantes de cet algorithme, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux noeuds. Chacun de ces 3 parcours définissent un ordre implicite (préfixé, infixé, postfixé) sur l'affichage et le traitement des données contenues dans l'arbre.

Algorithme de parcours en pré-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅ alors
   Traiter-1 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Algorithme de parcours en post-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅ alors
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```

Algorithme de parcours en ordre symétrique :

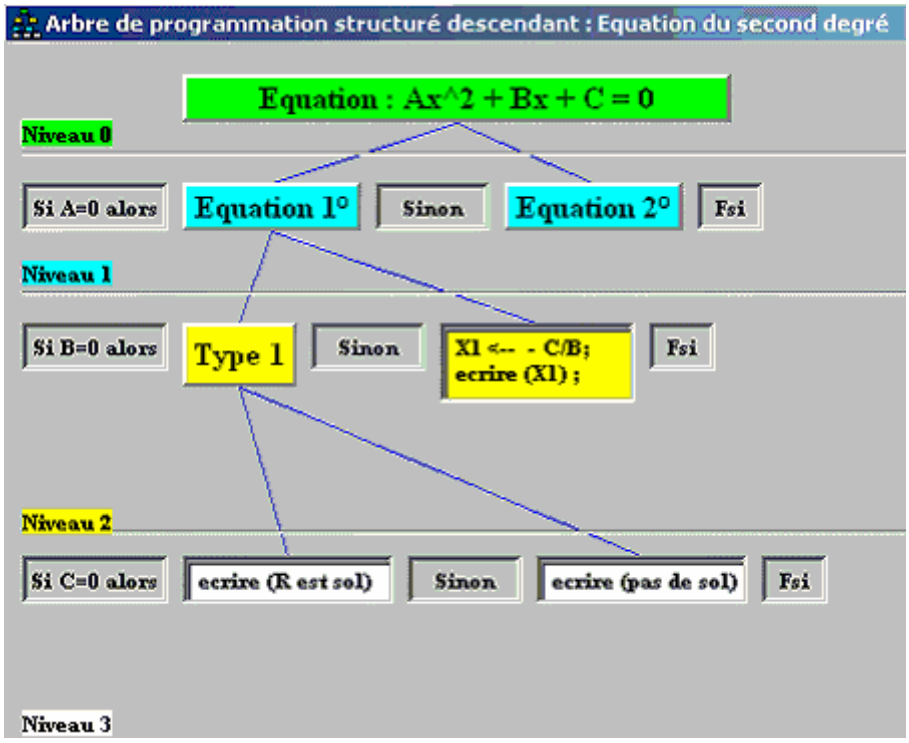
```
parcourir ( Arbre )
si Arbre ≠ ∅ alors
   parcourir ( Arbre.filsG) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Illustration pratique d'un parcours général en profondeur

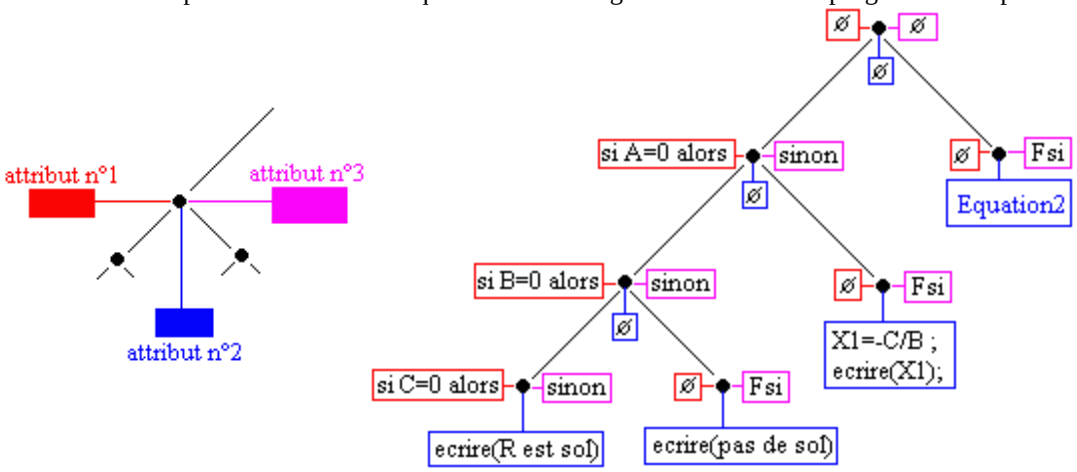
Le lecteur trouvera plus loin des exemples de parcours selon l'un des 3 ordres infixé, préfixé, postfixé, nous proposons ici un exemple didactique de parcours général avec les 3 traitements.

Nous allons voir comment utiliser une telle structure arborescente afin de restituer du texte algorithmique linéaire en effectuant un parcours en profondeur.

Voici ce que nous donne une analyse descendante du problème de résolution de l'équation du second degré (nous avons fait figurer uniquement la branche gauche de l'arbre de programmation) :



Ci-dessous une représentation schématique de la branche gauche de l'arbre de programmation précédent :

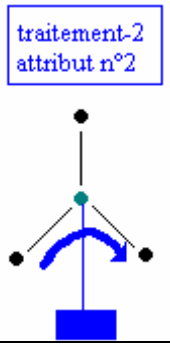
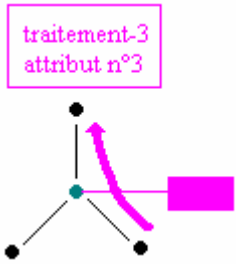


Nous avons établi un modèle d'arbre (binaire ici) où les informations au noeud sont au nombre de 3 (nous les nommerons attribut n°1, attribut n°2 et attribut n°3). Chaque attribut est une **chaîne de caractères**, vide s'il y a lieu.

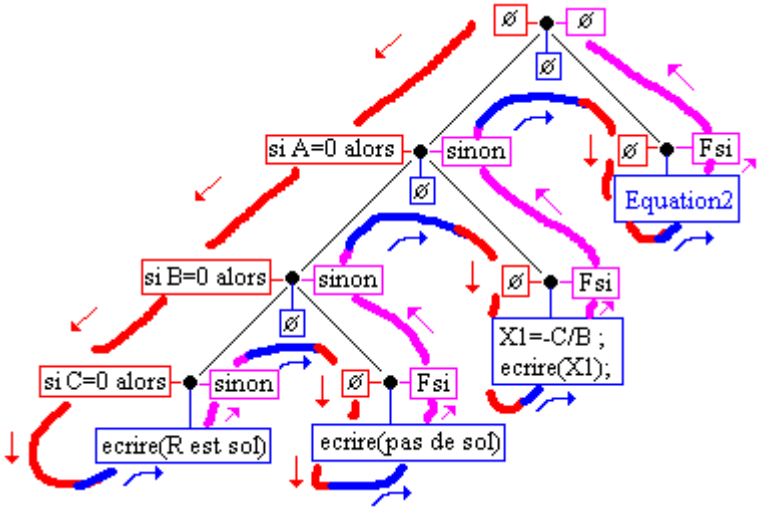
Nous noterons ainsi un attribut contenant une chaîne vide : ∅

Traitement des attributs pour produire le texte

traitement-1 attribut n°1	Traiter-1 (**Arbre.Racine.Attribut n°1**) consiste à écrire le contenu de l'Attribut n°1 : **si** Attribut n°1 non vide **alors** ecrire(Attribut n°1) **Fsi**

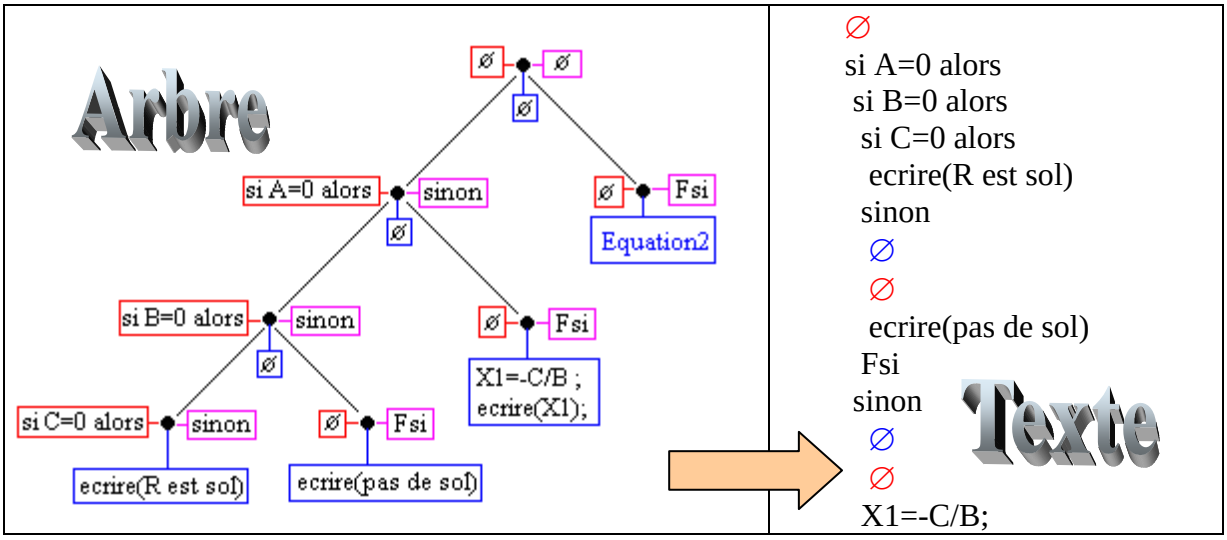
	Traiter-2 (**Arbre.Racine.Attribut n°2**) consiste à écrire le contenu de l'Attribut n°2 : **si** Attribut n°2 non vide **alors** ecrire(Attribut n°2) **Fsi**
	Traiter-3 (**Arbre.Racine.Attribut n°3**) consiste à écrire le contenu de l'Attribut n°3 : **si** Attribut n°3 non vide **alors** ecrire(Attribut n°3) **Fsi**

Parcours en profondeur de l'arbre de programmation de l'équation du second degré :



parcourir (Arbre)

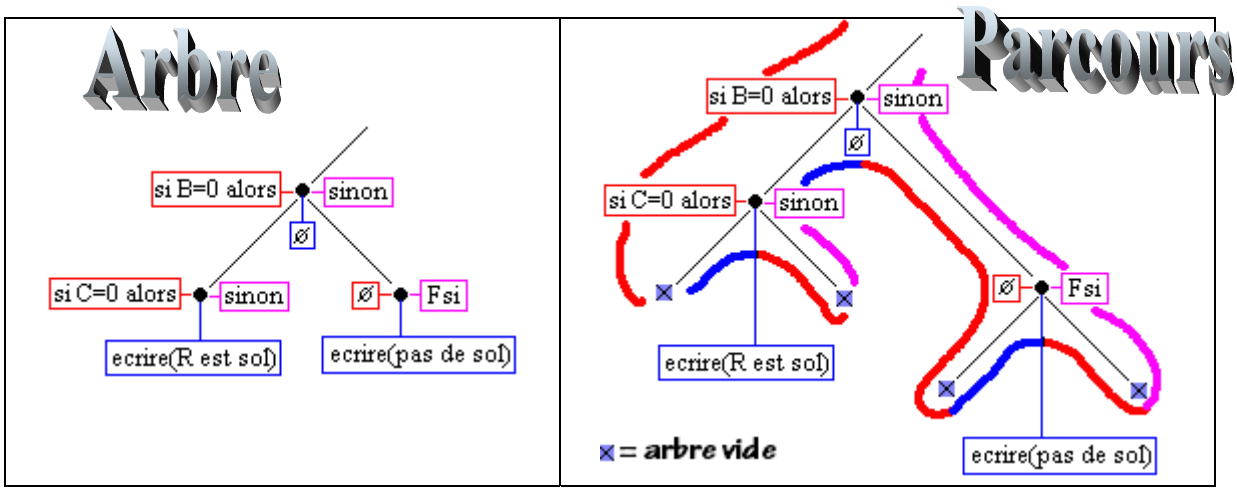
si Arbre ≠ ∅ alors
  Traiter-1 (**Attribut n°1**) ;
  parcourir (Arbre.filsG) ;
  Traiter-2 (**Attribut n°2**) ;
  parcourir (Arbre.filsD) ;
  Traiter-3 (**Attribut n°3**) ;
Fsi



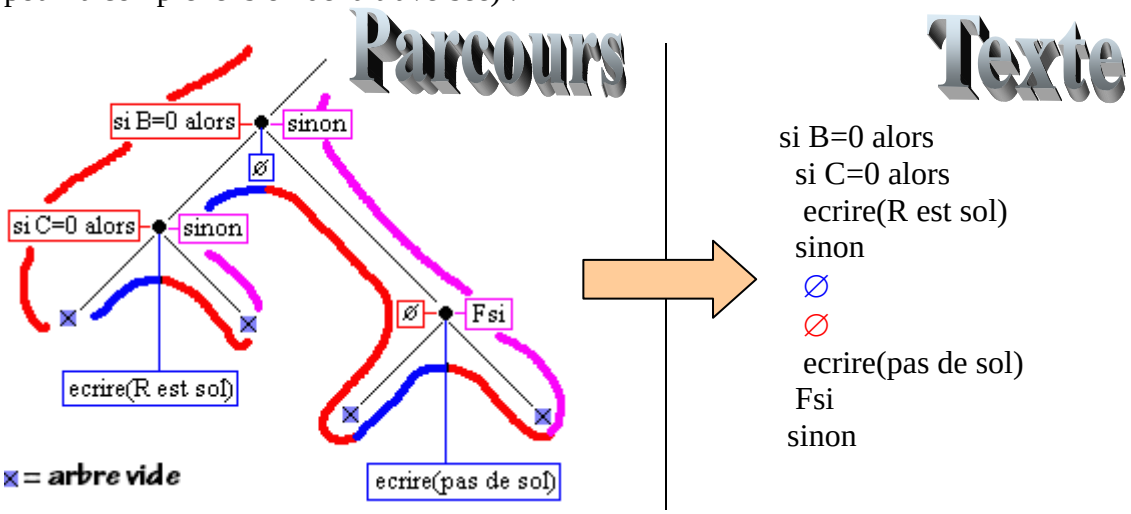
si Arbre ≠ ∅ alors Traiter-1 (**Attribut n°1**) ; **parcourir (Arbre.filsG) ;** Traiter-2 (**Attribut n°2**) ; **parcourir (Arbre.filsD) ;** Traiter-3 (**Attribut n°3**) ; **Fsi**	ecrire(X1); Fsi sinon ∅ ∅ Equation2 Fsi ∅

Rappelons que le symbole ∅ représente la chaîne vide il est uniquement mis dans le texe dans le but de permettre le suivi du parcours de l'arbre.

Pour bien comprendre le parcours aux feuilles de l'arbre précédent, nous avons fait figurer ci-dessous sur un exemple, les **noeuds vides** de chaque feuille et le **parcours complet associé** :

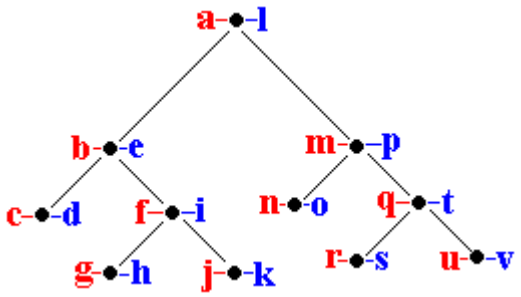


Le parcours partiel ci-haut produit le texte algorithmique suivant (le symbole ∅ est encore écrit pour la compréhension de la traversée) :



Exercice

Soit l'arbre ci-contre possédant 2 attributs par noeuds (un symbole de type caractère)



On propose le traitement en profondeur de l'arbre comme suit : L'attribut de gauche est écrit en descendant, l'attribut de droite est écrit en remontant, il n'y a pas d'attribut ni de traitement lors de l'examen du fils droit en venant du fils gauche. écrire la chaîne de caractère obtenue par le parcours ainsi défini.	**Réponse :** **abcdfghjkiemnoqrsuvtpl**

Terminons cette revue des descriptions algorithmiques des différents parcours classiques d'arbre binaire avec le parcours en largeur (*Cet algorithme nécessite l'utilisation d'une file du type Fifo dans laquelle l'on stocke les nœuds*)**.**

Algorithme de parcours en largeur

```
Largeur ( Arbre )

si Arbre ≠ ∅  alors
  ajouter racine de l'Arbre dans Fifo;
  tantque Fifo ≠ ∅ faire
    prendre premier de Fifo;
    traiter premier de Fifo;
    si  filsG de premier de Fifo ≠ ∅  alors
      ajouter filsG de premier de Fifo dans Fifo;
    Fsi
    si  filsD de premier de Fifo ≠ ∅  alors
      ajouter filsD de premier de Fifo dans Fifo;
    Fsi
  ftant
Fsi
```

2.6 Insertion, suppression, recherche dans un arbre binaire de recherche

Algorithme d'insertion dans un arbre binaire de recherche

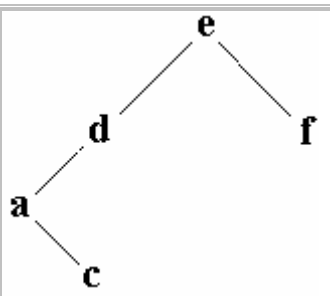
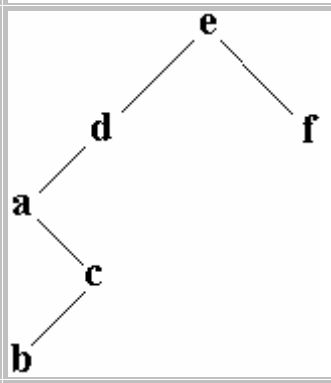
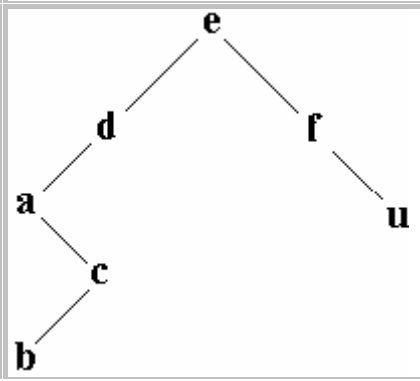
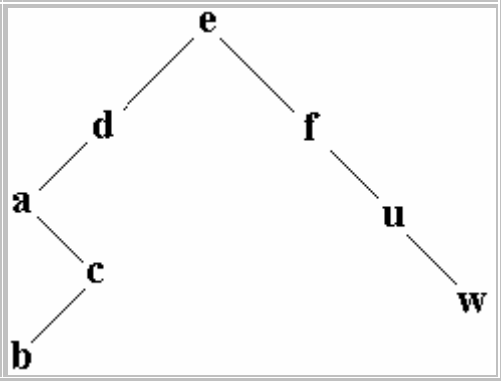
*placer l'élément Elt dans l'arbre **Arbre** par adjonctions successives aux feuilles*

placer (Arbre Elt **)**

si Arbre = ∅ **alors**
  creer un nouveau noeud contenant Elt **;**
  Arbre.Racine = ce nouveau noeud
sinon
{ - tous les éléments "info" de tous les noeuds du sous-arbre de gauche sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)
- tous les éléments "info" de tous les noeuds du sous-arbre de droite sont supérieurs à l'élément "info" du noeud en cours (arbre)
}
  si clef (Elt) ≤ clef (**Arbre.Racine**) **alors**
    placer (Arbre.filsG Elt **)**
  sinon
    placer (Arbre.filsD Elt **)**
Fsi

Soit par exemple la liste de caractères alphabétiques : **e d f a c b u w** *, que nous rangeons dans cet ordre d'entrée dans un arbre binaire de recherche. Ci-dessous le suivi de l'algorithme de placements successifs de chaque caractère de cette liste dans un arbre de recherche:*

Insertions successives des éléments	Arbre de recherche obtenu
placer (racine , '**e**') *e est la racine de l'arbre.*	e
placer (racine , '**d**') *d < e donc fils gauche de e.*	e d
placer (racine , '**f**') *f > e donc fils droit de e.*	e d f
placer (racine , '**a**') *a < e donc à gauche, a < d donc fils gauche de d.*	e d f a

placer (racine , '**c**') *c < e donc à gauche, c < d donc à gauche, c > a donc fils droit de a.*	
placer (racine , '**b**') *b < e donc à gauche, b < d donc à gauche, b > a donc à droite de a, b < c donc fils gauche de c.*	
placer (racine , '**u**') *u > e donc à droite de e, u > f donc fils droit de f.*	
placer (racine , '**w**') *w > e donc à droite de e, w > f donc à droite de f, w > u donc fils droit de u.*	

Algorithme de recherche dans un arbre binaire de recherche

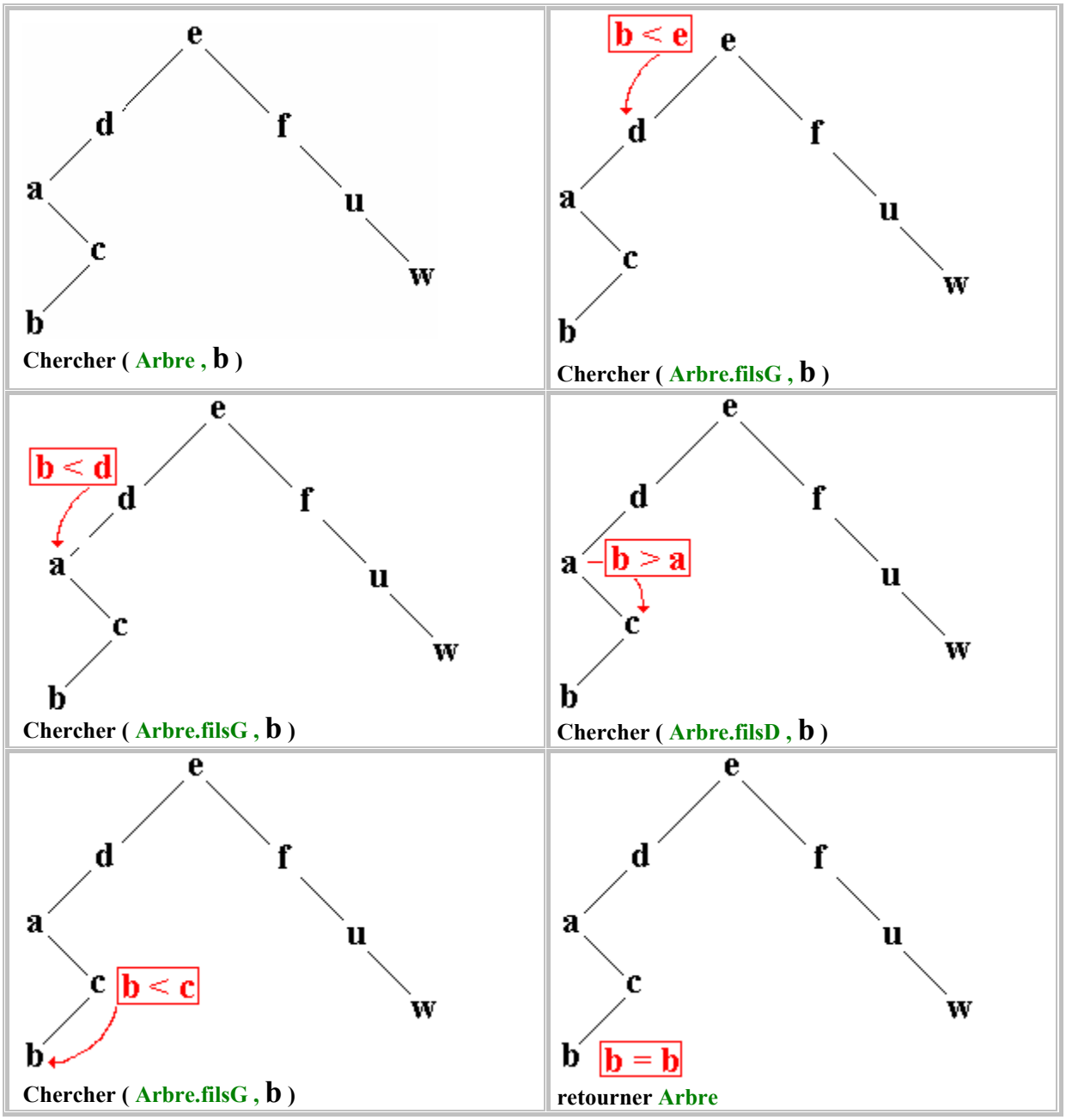
*chercher l'élément Elt dans l'arbre **Arbre :***

Chercher (Arbre Elt **) : Arbre**

si Arbre = ∅ **alors**
  Afficher Elt **non trouvé dans l'arbre;**
sinon
  si clef (Elt) < clef (**Arbre.Racine**) **alors**

```
    Chercher ( Arbre.filsG Elt ) //on cherche à gauche
  sinon
     si clef ( Elt )  >  clef ( Arbre.Racine ) alors
       Chercher ( Arbre.filsD Elt ) //on cherche à droite
     sinon retourner Arbre.Racine  //l'élément est dans ce noeud
     Fsi
  Fsi
Fsi
```

Ci-dessous le suivi de l'algorithme de recherche du caractère **b** *dans l'arbre précédent :*



Algorithme de suppression dans un arbre binaire de recherche

Afin de pouvoir supprimer un élément dans un arbre binaire de recherche, il est nécessaire de pouvoir d'abord le localiser, ensuite supprimer le noeud ainsi trouvé et éventuellement procéder à la réorganisation de l'arbre de recherche.

Nous supposons que notre arbre binaire de recherche ne possède que des éléments tous distincts (pas de redondance).

```
  supprimer l'élément Elt dans l'arbre Arbre :
Supprimer ( Arbre , Elt ) : Arbre
 local noeudMax : Noeud
si Arbre = ∅ alors
    Afficher Elt non trouvé dans l'arbre;
sinon
   si clef ( Elt )  < clef ( Arbre.Racine )   alors
         Supprimer ( Arbre.filsG , Elt ) //on cherche à gauche
   sinon
    si clef ( Elt )  >  clef ( Arbre.Racine ) alors
         Supprimer ( Arbre.filsD , Elt ) //on cherche à droite
    sinon //l'élément est dans ce noeud
      si Arbre.filsG = ∅ et Arbre.filsD ≠ ∅ alors //sous-arbre gauche vide
         remplacer Arbre par son sous-arbre droit Arbre.filsD
      sinon
        si Arbre.filsD = ∅  et Arbre.filsG ≠ ∅ alors  //sous-arbre droit vide
          remplacer Arbre par son sous-arbre gauche Arbre.filsG
        sinon
          si Arbre.filsD ≠ ∅  et Arbre.filsG ≠ ∅ alors  //le noeud a deux descendants
            noeudMax ← maxClef( Arbre.filsG ); //noeudMax = le max du fils gauche
            clef ( Arbre.Racine ) ← clef (noeudMax ); //remplacer etiquette
            Supprimer ( Arbre.filsG , clef (noeudMax ) ) //on cherche à gauche
          sinon //le nœud est alors une feuille
            détruire (Arbre)
          Fsi
        Fsi
      Fsi
    Fsi
  Fsi
Fsi
```

Cet algorithme utilise l'algorithme récursif **maxClef** *de recherche de la plus grandeclef dans l'arbre* ***Arbre :***

//par construction il suffit de descendre systématiquement toujours le plus à droite

```
maxClef ( Arbre ) : Arbre
si Arbre.filsD = ∅ alors
  retourner Arbre.Racine //c'est le plus grand élément
sinon
```

maxClef (Arbre.filsD)
Fsi

Classe générique C# d'arbre binaire avec parcours en profondeur

Code C# d'une classe générique d'arbre binaire ArbreBin<T0> avec parcours en profondeur, traduit à partir des algorithmes précédents :

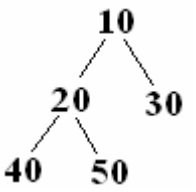
```
interface IArbreBin<T0>  {
   T0 Info { get;      set;       }
}

class ArbreBin<T0> : IArbreBin<T0>
{
   private T0 InfoLoc;
   private ArbreBin<T0> fg;
   private ArbreBin<T0> fd;

   public ArbreBin(T0 s) : this( )
   {
      InfoLoc = s;
   }

   public ArbreBin()
   {
      InfoLoc = default(T0);
      fg = default(ArbreBin<T0>);
      fd = default(ArbreBin<T0>);
   }

   public T0 Info
   {
      get { return InfoLoc; }
      set { InfoLoc = value; }
   }

   public ArbreBin<T0> filsG
   {
      get { return this.fg; }
      set
      {
         if ( value != null )
            fg = new ArbreBin<T0>(value.Info);
         else
            fg = default(ArbreBin<T0>);
      }
   }

   public ArbreBin<T0> filsD
   {
      get { return this.fd; }
     set
      {
         if ( value != null )
            fd = new ArbreBin<T0>(value.Info);
         else
```

```
            fd = default(ArbreBin<T0>);
        }
    }

    public void prefixe()
    {
        Console.WriteLine(this.Info);
        if (fg != null)
            this.fg.prefixe();
        if (fd != null)
            this.fd.prefixe();
    }

    public void postfixe()
    {
        if (fg != null)
            this.fg.postfixe();
        if (fd != null)
            this.fd.postfixe();
        Console.WriteLine(this.Info);
    }

    public void infixe()
    {
        if (fg != null)
            this.fg.infixe();
        Console.WriteLine(this.Info);
        if (fd != null)
            this.fd.infixe();
    }
  }
```

Cette classe C# permet de construire et de parcourir en profondeur selon les trois parcours précédemment étudié. Ci-après un exemple d'utilisation de cette classe T0 = int :

```
static void Main(string[] args)
{
    ArbreBin<int> treeRac = new ArbreBin<int>();
    treeRac.Info = 10;
    treeRac.filsG = new ArbreBin<int>(20);
    treeRac.filsD = new ArbreBin<int>(30);
    treeRac.filsG.filsG = new ArbreBin<int>(40);
    treeRac.filsG.filsD = new ArbreBin<int>(50);
    Console.WriteLine(" treeRac.info = " + treeRac.Info);
    Console.WriteLine(" treeRac.filsG.info = " + treeRac.filsG.Info);
    Console.WriteLine(" treeRac.filsD.info = " + treeRac.filsD.Info);
    Console.WriteLine(" treeRac.filsG.filsG.info = "+treeRac.filsG.filsG.Info);
    Console.WriteLine("--- parcours prefixe ---");
        treeRac.prefixe();
    Console.WriteLine("--- parcours postfixe ---");
        treeRac.postfixe();
    Console.WriteLine("--- parcours infixe ---");
        treeRac.infixe();
    Console.ReadLine();
}
```

La méthode Main construit cet arbre binaire :



Les parcours en profondeur donnent les résultats suivants :

```
treeRac.info = 10
treeRac.filsG.info = 20
treeRac.filsD.info = 30
treeRac.filsG.filsG.info = 40
--- parcours prefixe ---
10
20
40
50
30
--- parcours postfixe ---
40
50
20
30
10
--- parcours infixe ---
40
20
50
10
30
```

Classe générique C# d'arbre binaire avec parcours en largeur

On ajoute à la classe précédente le parcours en largeur, pour cela nous déclarons deux nouveaux champs privés dans la classe pour représenter la FIFO et le premier élément de cette FIFO. Pour la file FIFO, nous utilisons la classe générique de file de .Net Queue<T0>, le type d'élément T0 étant alors ArbreBin<T> :

```
private Queue<ArbreBin<T>> Fifo = new Queue<ArbreBin<T>>();
private ArbreBin<T> Premier;
```

Algorithme	*Méthode C#*

Largeur (Arbre)
 	
si Arbre ≠ ∅ alors
ajouter racine de l'**Arbre** dans Fifo;
tantque Fifo ≠ ∅ **faire**
prendre premier de Fifo;
traiter premier de Fifo;
si filsG de premier de Fifo ≠ ∅ alors
ajouter **filsG** de premier de Fifo dans Fifo;
Fsi
si filsD de premier de Fifo ≠ ∅ alors
ajouter **filsD** de premier de Fifo dans Fifo;
Fsi
ftant
Fsi
	public void largeur()
	{
	Fifo.Enqueue(this);
	while (Fifo.Count != 0)
	{
	Premier = Fifo.Dequeue();
	Console.WriteLine(Premier.Info);
	if (Premier.filsG != null)
	Fifo.Enqueue(Premier.filsG);
	if (Premier.filsD != null)
	Fifo.Enqueue(Premier.filsD);
	}
	}

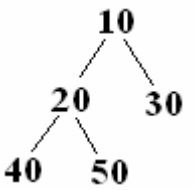
Voici uniquement le code rajouté relatif au parcours en largeur, dans la classe ArbreBin<T>, le reste est identique au code défini dans les pages précédentes :

```
class ArbreBin<T0> : IArbreBin<T0>
{
   private T0 InfoLoc;
   private ArbreBin<T0> fg;
   private ArbreBin<T0> fd;
   public ArbreBin(T0 s) : this( ) { …  }
   public ArbreBin() {… }
   public T0 Info   { …   }
   public ArbreBin<T0> filsG   { …  }
   public ArbreBin<T0> filsD { …  }
   public void prefixe()  { …  }
   public void postfixe() { … }
   public void infixe() { … }

   private Queue<ArbreBin<T>> Fifo = new Queue<ArbreBin<T>>();
   private ArbreBin<T> Premier;
   public void largeur()
  {
      Fifo.Enqueue(this);
      while (Fifo.Count != 0)
      {
         Premier = Fifo.Dequeue();
         Console.WriteLine(Premier.Info);
         if (Premier.filsG != null)
            Fifo.Enqueue(Premier.filsG);
         if (Premier.filsD != null)
            Fifo.Enqueue(Premier.filsD);
      }
   }
}
```

Ci-après, la reprise de l'exemple d'utilisation de cette classe avec T0 = int :

```
static void Main(string[ ] args)
{
    ArbreBin<int> treeRac = new ArbreBin<int>();
    treeRac.Info = 10;
    treeRac.filsG = new ArbreBin<int>(20);
    treeRac.filsD = new ArbreBin<int>(30);
    treeRac.filsG.filsG = new ArbreBin<int>(40);
    treeRac.filsG.filsD = new ArbreBin<int>(50);
    Console.WriteLine(" treeRac.info = " + treeRac.Info);
    Console.WriteLine(" treeRac.filsG.info = " + treeRac.filsG.Info);
    Console.WriteLine(" treeRac.filsD.info = " + treeRac.filsD.Info);
    Console.WriteLine(" treeRac.filsG.filsG.info = "+treeRac.filsG.filsG.Info);
    Console.WriteLine("--- parcours prefixe ---");
          treeRac.prefixe();
    Console.WriteLine("--- parcours postfixe ---");
          treeRac.postfixe();
    Console.WriteLine("--- parcours infixe ---");
          treeRac.infixe();
    Console.WriteLine("--- parcours Largeur ---");
          treeRac.largeur();
    Console.ReadLine();
}
```

La méthode Main construit cet arbre binaire :



Les parcours en profondeur donnent les résultats suivants :

Le parcours en largeur donne le résultat suivant :

```
treeRac.info = 10
treeRac.filsG.info = 20
treeRac.filsD.info = 30
treeRac.filsG.filsG.info = 40
--- parcours prefixe ---
10
20
40
50
30
--- parcours postfixe ---
40
50
20
30
10
--- parcours infixe ---
40
20
50
10
30
--- parcours Largeur ---
10
20
30
40
50
```

Classe générique C# d'arbre binaire de recherche

On définit une nouvelle classe d'arbre binaire de recherche générique que nous nommons ArbreBinRech<T> dont chaque nœud est un arbre binaire de type ArbreBin<T>; on contraint le type T à implémenter l'interface IComparable afin de pouvoir travailler sur des données possédant une méthode CompareTo autorisant un test de comparaison de leurs valeurs :

1°) Méthode placer pour l'insertion dans un arbre binaire de recherche :

```
class ArbreBinRech<T>  where T : IComparable<T>
{
    public ArbreBin<T> racine;

    public ArbreBinRech()
    {
      racine = null;
    }
    public ArbreBinRech(ArbreBin<T> tree)
    {
      racine = tree;
    }
```

Méthodes C#	*Algorithme - rappel*

`public void placer(T elt)`
`{`
`  if (racine == null)`
`  {`
`  racine = new ArbreBin<T>(elt);`
`  return;`
`  }`
`  placerR(racine, elt);`
`}`
	placer (Arbre Elt **)**
	
	si Arbre = ∅ **alors**
	creer un nouveau noeud contenant Elt **;**
	Arbre.Racine = ce nouveau noeud
	sinon
	{ - tous les éléments "info" de tous les noeuds du sous-arbre de gauche

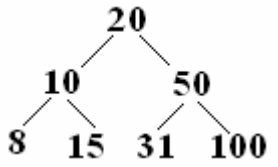
private void placerR(ArbreBin<T> tree, T elt)
{
if (elt.CompareTo(tree.Info) <= 0)
{
if (tree.filsG == null)
tree.filsG = new ArbreBin<T>(elt);
else
this.placerR(tree.filsG, elt);
}
else
{
if (tree.filsD == null)
tree.filsD = new ArbreBin<T>(elt);
else
this.placerR(tree.filsD, elt);
}
}
	sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)
	- tous les éléments "info" de tous les noeuds du sous-arbre de droite sont supérieurs à l'élément "info" du noeud en cours (arbre)
	}
	si clef (Elt) ≤ clef (**Arbre.Racine**) **alors**
	placer (Arbre.filsG Elt **)**
	sinon
	placer (Arbre.filsD Elt **)**
	Fsi

}

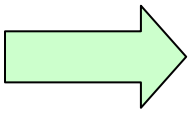
Ci-après, exemple d'utilisation de cette classe avec T = **int** qui implémente **IComparable** :

```
static void Main(string[ ] args)
  {
    ArbreBinRech<int> treeRech = new ArbreBinRech<int>();
    treeRech.placer(20);
    treeRech.placer(50);
    treeRech.placer(100);
    treeRech.placer(10);
    treeRech.placer(15);
    treeRech.placer(8);
    treeRech.placer(31);
    Console.WriteLine("--- arbre recherche en Largeur ---");
    treeRech.racine.largeur();
    Console.WriteLine("--- arbre recherche en prefixe ---");
    treeRech.racine.prefixe();
    Console.WriteLine("--- arbre recherche en postfixe ---");
    treeRech.racine.postfixe();
    Console.WriteLine("--- arbre recherche en infixe ---");
    treeRech.racine.infixe();
    Console.Read();
  }
```

La méthode Main précédente construit l'arbre binaire de recherche qui suit :



Les différents parcours donnent les résultats suivants :



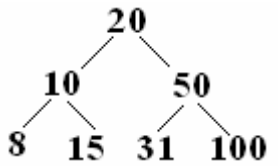
```
--- arbre recherche en Largeur ---
20
10
50
8
15
31
100
--- arbre recherche en prefixe ---
20
10
8
15
50
31
100
--- arbre recherche en postfixe ---
8
15
10
31
100
50
20
--- arbre recherche en infixe ---
8
10
15
20
31
50
100
```

2°) Méthode chercher pour la recherche dans un arbre binaire de recherche :

Méthodes C#	*Algorithme - rappel*

`public ArbreBin<T> rechercher(T clef)`
`{`
`  return rechercherR(racine, clef);`
`}`
 	
`private ArbreBin<T> rechercherR(ArbreBin<T> tree, T clef)`
`{`
`  if (tree == null)`
`  return null;`
`  if (clef.CompareTo(tree.Info) == 0)`
`  return tree;`
`  if (clef.CompareTo(tree.Info) < 0)`
`  return rechercherR(tree.filsG, clef);`
`  else`
`  return rechercherR(tree.filsD, clef);`
`}`
	*chercher l'élément Elt dans l'arbre **Arbre :***
	Chercher (Arbre Elt) : Arbre
	si Arbre = ∅ alors
	Afficher Elt **non trouvé dans l'arbre;**
	sinon
	si clef (Elt) < clef (**Arbre.Racine**) **alors**
	Chercher (Arbre.filsG Elt) ***//on cherche à gauche***
	sinon
	si clef (Elt) > clef (**Arbre.Racine**) **alors**
	Chercher (Arbre.filsD Elt) ***//on cherche à droite***
	sinon retourner Arbre.Racine ***//l'élément est dans ce noeud***
	Fsi
	Fsi
	Fsi

On utilise l'arbre binaire de recherche précédent :

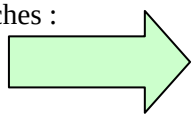


Dans la méthode Main précédente on ajoute le code suivant permettant de rechercher les nœuds de clefs fixées (31 et 78) :

```
Console.WriteLine("--- recherche de clef : 31 ---");
if (treeRech.rechercher(31) == null)
   Console.WriteLine(">>> 31 pas trouvée.");
else
   Console.WriteLine(">>> " + treeRech.rechercher(31).Info);
```

```
Console.WriteLine("--- recherche de clef : 78 ---");
if (treeRech.rechercher(78) == null)
   Console.WriteLine(">>> 78 pas trouvée.");
else
   Console.WriteLine(">>> " + treeRech.rechercher(78).Info);
```

Résultats des recherches :



```
--- recherche de clef : 31 ---
>>> 31
--- recherche de clef : 78 ---
>>> 78 pas trouvée.
```

3°) Méthode supprimer pour la suppression dans un arbre binaire de recherche :

Méthodes C#	*Algorithme - rappel*

`public void supprimer(T clef)`
`{`
`  if (racine != null)`
`  supprimerR(racine, null, 'X', clef);`
`}`
 	
`private void supprimerR(ArbreBin<T> tree,`
`  ArbreBin<T> parent, char branche, T clef)`
`{`
`  ArbreBin<T> noeudMax;`
`  if (tree == null)`
`  return;//clef non trouvée`
`  else`
`  if (clef.CompareTo(tree.Info) < 0)`
`  {`
`  //on cherche à gauche :`
`  this.supprimerR(tree.filsG, tree, 'G', clef);`
`  }`
`  else`
`  if (clef.CompareTo(tree.Info) > 0)`
`  {`
`  //on cherche à droite :`
`  this.supprimerR(tree.filsD, tree, 'D', clef);`
`  }`
`  else //l'élément est dans ce nœud:`
`  if (tree.filsD == null && tree.filsG != null)`
`  {//sous-arbre droit vide`
`  //remplacer arbre par son sous-arbre gauche :`
`  tree.Info = tree.filsG.Info;`
`  tree.filsD = tree.filsG.filsD;`
`  tree.filsG = tree.filsG.filsG;`
`  }`
`  else`
`  if (tree.filsG == null && tree.filsD != null)`
`  {//sous-arbre gauche vide`
`  //remplacer arbre par son sous-arbre droit :`
`  tree.Info = tree.filsD.Info;`
`  tree.filsD = tree.filsD.filsD;`
`  tree.filsG = tree.filsD.filsG;`
`  }`
`  else`
`  if (tree.filsG != null && tree.filsD != null)`
`  {// le noeud a 2 fils`
`  //le max du fils gauche :`
	Cette méthode suit très exactement l'algorithme proposé plus haut.
	
	La méthode supprimerR possède comme dans les deux traitements précédents les deux paramètres ArbreBin<T> tree et T clef, elle possède en plus 2 autres paramètres :
	• ArbreBin<T> parent = la réfernce du raent du nœud actuel.
	• char branche = un caractère 'G' ou 'D' indiquant si le nœud actuel est un filsG ou un filsD du parent.
	
	L'implantation du remplacement d'un nœud par son fils gauche s'effectue ainsi :
	
	tree.Info = tree.filsG.Info;
	tree.filsD = tree.filsG.filsD;
	tree.filsG = tree.filsG.filsG;
	
	L'implantation du remplacement d'un nœud par son fils droit s'effectue ainsi :
	
	tree.Info = tree.filsD.Info;
	tree.filsD = tree.filsD.filsD;
	tree.filsG = tree.filsD.filsG;

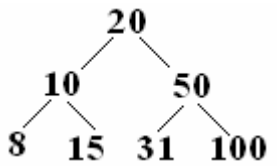
```
                noeudMax = maxClef(tree.filsG);
                tree.Info = noeudMax.Info; //← remplacer clef
                //on cherche à gauche :
                this.supprimerR(tree.filsG,
                      tree, 'G', noeudMax.Info);
               }
               else // le noeud est une feuille:  on le détruit
               {
                  if (branche == 'D')
                     parent.filsD = null;
                  else
                    if (branche == 'G')
                       parent.filsG = null;
               }
}

private ArbreBin<T> maxClef(ArbreBin<T> tree)
{
  if (tree.filsD == null)
     return tree;
  else
     return maxClef(tree.filsD);
}
```

L'implantation de la destruction du nœud contenant la clef est effectuée par la suppression dans le parent de la référence pointant vers ce neoud.

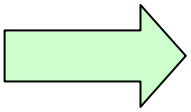
Le caractère "branche" permet de savoir si ce nœud est le fils gauche (branche == 'G') ou bien le fils droit du parent (branche == 'D').

fils gauche => parent.filsG = null;
fils droit => parent.filsD = null;

On utilise l'arbre binaire de recherche précédent :



Résultats de diverses suppressions :



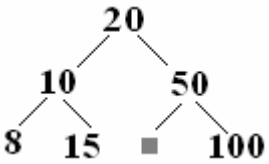
Dans la méthode Main précédente on ajoute le code suivant permettant de rechercher les nœuds de clefs fixées (31 et 78) :

```
int clef = 20;
Console.WriteLine("--- suppression clef : " + clef + "  ---");
treeRech.supprimer(clef);
treeRech.racine.largeur( );
```

clef = 31

```
--- suppression clef : 31
20
10
50
8
15
100
```

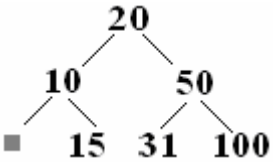
arbre après suppression :



clef = 8

```
--- suppression clef : 8
20
10
50
15
31
100
```

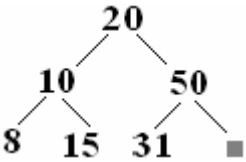
arbre après suppression :



clef = 100

```
--- suppression clef : 100
20
10
50
8
15
31
```

arbre après suppression :

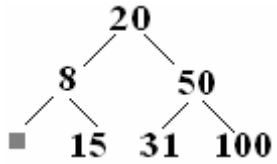


Résultats de diverses suppressions :

clef = 10

```
--- suppression clef : 10
20
8
50
15
31
100
```

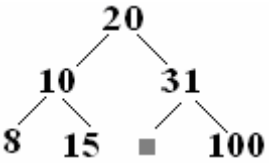
arbre après suppression :



clef = 50

```
--- suppression clef : 50
20
10
31
8
15
100
```

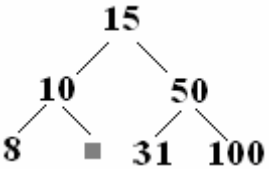
arbre après suppression :



clef = 20

```
--- suppression clef : 20
15
10
50
8
31
100
```

arbre après suppression :



Principes des bases de données

Plan général:

1. Introduction et Généralités
2. Le modèle de données relationnelles
3. Principes fondamentaux d'une algèbre relationnelle
4. SQL et Algèbre relationnelle

1. Introduction et Généralités

1.1 Notion de système d'information

L'informatique est une science du traitement de l'information, laquelle est représentée par des données Aussi, très tôt, on s'est intéressé aux diverses manières de pouvoir stocker des données dans des mémoires auxiliaires autres que la mémoire centrale. Les données sont stockées dans des périphériques dont les supports physiques ont évolué dans le temps : entre autres, d'abord des cartes perforées, des bandes magnétiques, des cartes magnétiques, des mémoires à bulles magnétiques, puis aujourd'hui des disques magnétiques, ou des CD-ROM ou des DVD.

La notion de fichier est apparue en premier : le fichier regroupe tout d'abord des objets de même nature, des enregistrements. Pour rendre facilement exploitables les données d'un fichier, on a pensé à différentes méthodes d'accès (accès séquentiel, direct, indexé).

Toute application qui gère des systèmes physiques doit disposer de paramètres sémantiques décrivant ces systèmes afin de pouvoir en faire des traitements. Dans des systèmes de gestion de clients les paramètres sont très nombreux (noms, prénoms, adresse, n°Sécu, sport favori, est satisfait ou pas,..) et divers (alphabétiques, numériques, booléens, ...).

Dès que la quantité de données est très importante, les fichiers montrent leurs limites et il a fallu trouver un moyen de stocker ces données et de les organiser d'une manière qui soit facilement accessible.

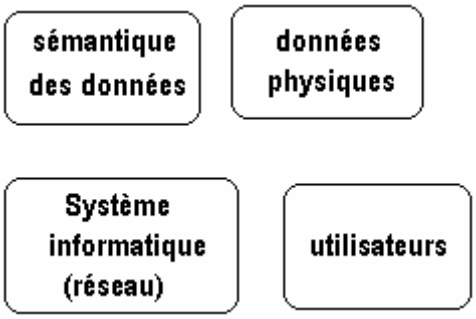
Base de données (BD)

Une BD est composée de données stockées dans des mémoires de masse sous une forme structurée, et accessibles par des applications différentes et des utilisateurs différents. Une BD doit pouvoir être utilisée par plusieurs utilisateurs en "même temps".

Une base de données est structurée par définition, mais sa structuration doit avoir un caractère universel : il ne faut pas que cette structure soit adaptée à une application particulière, mais qu'elle puisse être utilisable par plusieurs applications distinctes. En effet, un même ensemble de données peut être commun à plusieurs systèmes de traitement dans un problème physique (par exemple la liste des passagers d'un avion, stockée dans une base de données, peut aussi servir au service de police à vérifier l'identité des personnes interdites de séjour, et au service des douanes pour associer des bagages aux personnes….).

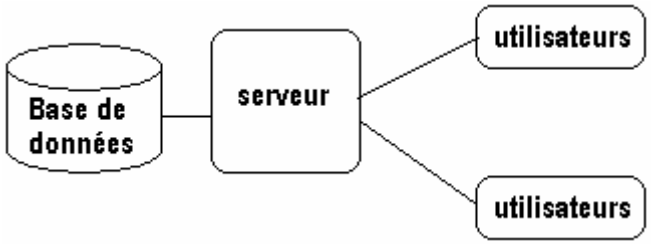
Système d'information

Dans une entreprise ou une administration, la structure sémantique des données, leur organisation logique et physique, le partage et l'accès à de grandes quantités de données grâce à un système informatique, se nomme un système d'information.



Les entités qui composent un système d'information

L'organisation d'un SI relève plus de la gestion que de l'informatique et n'a pas exactement sa place dans un document sur la programmation. En revanche la cheville ouvrière d'un système d'information est un outil informatique appelé un **SGBD** (système de gestion de base de données) qui repose essentiellement sur un système informatique composé traditionnellement d'une **BD** et d'un réseau de postes de travail consultant ou mettant à jour les informations contenues dans la base de données, elle-même généralement située sur un ordinateur-serveur.



Système de Gestion de Base de Données (SGBD)

Un SGBD est un ensemble de logiciels chargés d'assurer les fonctions minimales suivantes :
- Le maintien de la cohérence des données entre elles,
- le contrôle d'intégrité des données accédées,
- les autorisations d'accès aux données,
- les opérations classiques sur les données (consultation, insertion , modification, suppression)

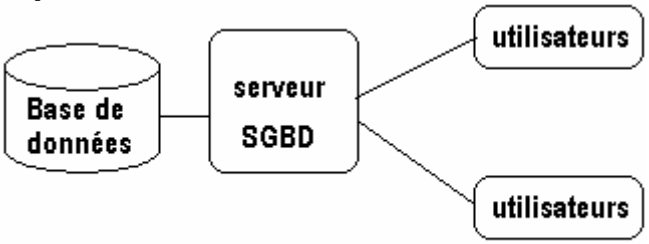
La cohérence des données est subordonnée à la définition de contraintes d'intégrité qui sont des règles que doivent satisfaire les données pour être acceptées dans la base. Les contraintes d'intégrité sont contrôlées par le moteur du SGBD :

- au niveau de chaque champ, par exemple le : prix est un nombre positif, la date de naissance est obligatoire.
- Au niveau de chaque table - voir plus loin la notion de clef primaire : deux personnes ne doivent pas avoir à la fois le même nom et le même prénom.
- Au niveau des relations entre les tables : contraintes d'intégrité référentielles.

Par contre la redondance des données (formes normales) **n'est absolument pas vérifiée automatiquement** par les SGBD, il faut faire des requêtes spécifiques de recherche d'anomalies (dites post-mortem) **à postériori**, ce qui semble être une grosse lacune de ces systèmes puisque les erreurs sont déjà présentes dans la base !

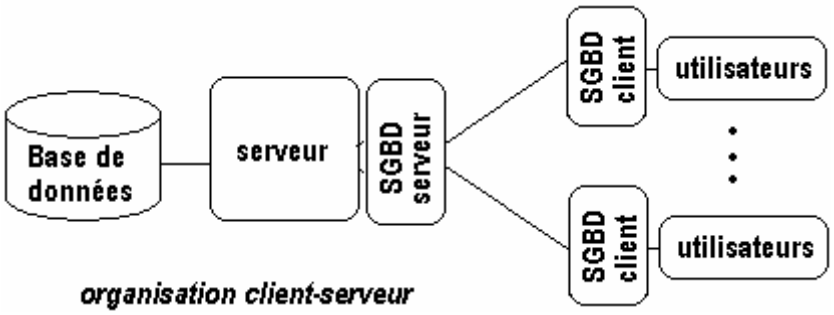
On organise actuellement les SGBD selon deux modes :

L'organisation locale selon laquelle le SGBD réside sur la machine où se trouve la base de données :



organisation locale

L'organisation client-serveur selon laquelle sur le SGBD est réparti entre la machine serveur locale supportant la BD (partie SGBD serveur) et les machines des utilisateurs (partie SGBD client). Ce sont ces deux parties du SGBD qui communiquent entre elles pour assurer les transactions de données :



Le caractère généraliste de la structuration des données induit une description abstraite de l'objet BD (Base de données). Les applications étant indépendantes des données, ces dernières peuvent donc être manipulées et changées indépendamment du programme qui y accédera en implantant les méthodes générales d'accès aux données de la base, conformément à sa structuration abstraite.

Une Base de Données peut être décrite de plusieurs points de vue, selon que l'on se place du côté de l'utilisateur ou bien du côté du stockage dans le disque dur du serveur ou encore du concepteur de la base.

Il est admis de nos jours qu'une **BD** est décrite en trois niveaux d'abstraction : un seul niveau a une existence matérielle physique et les deux autres niveaux sont une explication abstraite de ce niveau matériel.

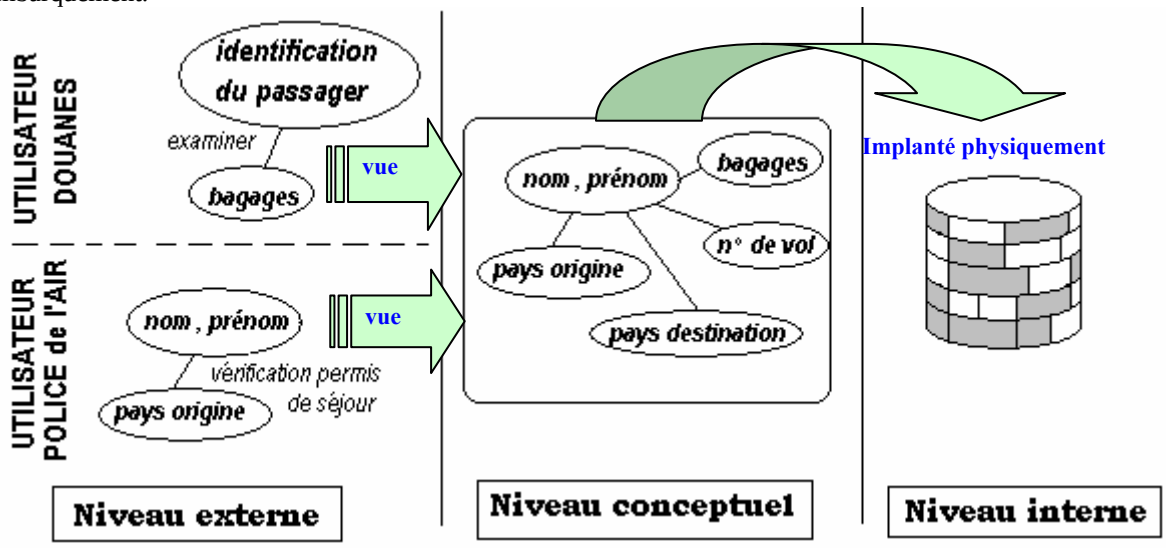
Les 3 niveaux d'abstraction définis par l'ANSI depuis 1975

- **Niveau externe** : correspond à ce que l'on appelle une vue de la BD ou la façon dont sont perçues au niveau de l'utilisateur les données manipulées par une certaine application (vue abstraite sous forme de schémas)

- **Niveau conceptuel** : correspond à la description abstraite des composants et des processus entrant dans la mise en œuvre de la BD. Le niveau conceptuel est le plus important car il est le résultat de la traduction de la description du monde réel à l'aide d'expressions et de schémas conformes à un modèle de définition des données.

- **Niveau interne** : correspond à la description informatique du stockage physique des données (fichiers séquentiels, indexages, tables de hachage,…) sur le disque dur.

Figurons pour l'exemple des passagers d'un avion, stockés dans une base de données de la compagnie aérienne, sachant qu'en plus du personnel de la compagnie qui a une vue externe commerciale sur les passagers, le service des douanes peut accéder à un passager et à ses bagages et la police de l'air peut accéder à un passager et à son pays d''embarquement.

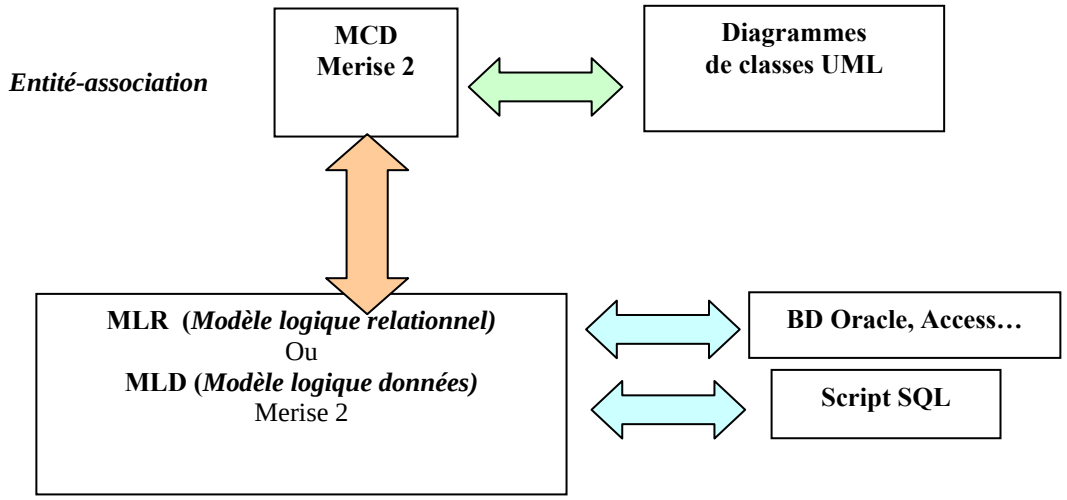


Le niveau conceptuel forme l'élément essentiel d'une BD et donc d'un SGBD chargé de gérer une BD, il est décrit avec un modèle de conception de données MCD avec la méthode française Merise qui est très largement répandu, ou bien par le formalisme des diagrammes de classes UML qui prend une part de plus en plus grande dans le formalisme de description conceptuelle des données (rappelons qu'UML est un langage de modélisation formelle, orienté objet et graphique ; Merise2 a intégré dans Merise ces concepts mais ne semble pas beaucoup être utilisé). Nous renvoyons le lecteur intéressé par cette partie aux très nombreux ouvrages écrits sur Merise ou sur UML.

Dans la pratique actuelle les logiciels de conception de BD intègrent à la fois la méthode Merise 2 et les diagrammes de classes UML. Ceci leur permet surtout la génération automatique et semi-automatique (paramétrable) de la BD à partir du modèle conceptuel sous forme de scripts (programmes simples) SQL adaptés aux différents SGBD du marché (ORACLE, SYBASE, MS-SQLSERVER,…) et les différentes versions de la BD ACCESS.

Les logiciels de conception actuels permettent aussi la rétro-génération (ou reverse engeneering) du modèle à partir d'une BD existante, cette fonctionnalité est très utile pour reprendre un travail mal documenté.

En résumé pratique :

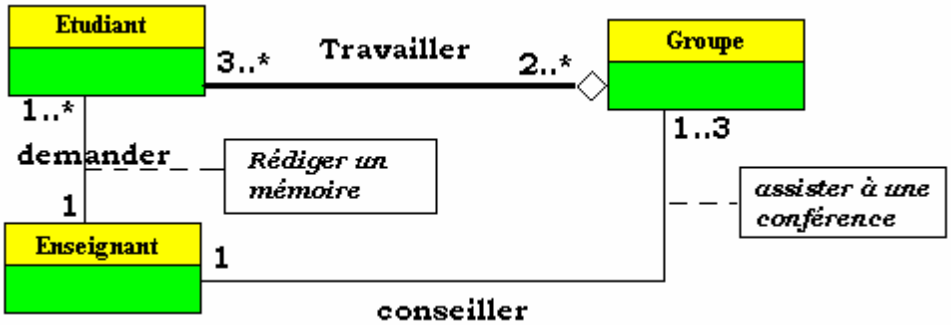


C'est en particulier le cas du logiciel français WIN-DESIGN dont une version démo est disponible à www.win-design.com et de son rival POWER-AMC (ex AMC-DESIGNOR).

Signalons enfin un petit logiciel plus modeste, très intéressant pour débuter avec version limitée seulement par la taille de l'exemple : CASE-STUDIO chez CHARONWARE. Les logiciels basés uniquement sur UML sont, à ce jour, essentiellement destinés à la génération de code source (Java, Delphi, VB, C++,…), les versions **Community** (versions logicielles libres) de ces logiciels ne permettent pas la génération de BD ni celle de scripts SQL.
Les quelques schémas qui illustreront ce chapitre seront décrits avec le langage UML.

L'exemple ci-après schématise en UML le mini-monde universitaire réel suivant :

- un enseignant pilote entre 1 et 3 groupes d'étudiants,
- un enseignant demande à 1 ou plusieurs étudiants de rédiger un mémoire,
- un enseignant peut conseiller aux groupes qu'il pilote d'aller assister à une conférence,
- un groupe est constitué d'au moins 3 étudiants,
- un étudiant doit s'inscrire à au moins 2 groupes.



Si le niveau conceptuel d'une BD est assis sur un modèle de conceptualisation de haut niveau (Merise, UML) des données, il est ensuite fondamentalement traduit dans le Modèle Logique de représentation des Données (MLD). Ce dernier s'implémentera selon un modèle physique des données.

Il existe plusieurs MLD **M**odèles **L**ogiques de **D**onnées et plusieurs modèles physiques, et pour un même MLD, on peut choisir entre plusieurs modèles physiques différents.

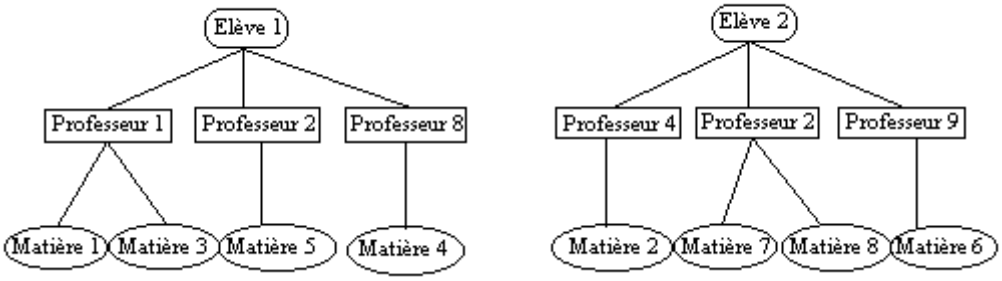
Il existe 5 grands modèles logiques pour décrire les bases de données.

Les modèles de données historiques

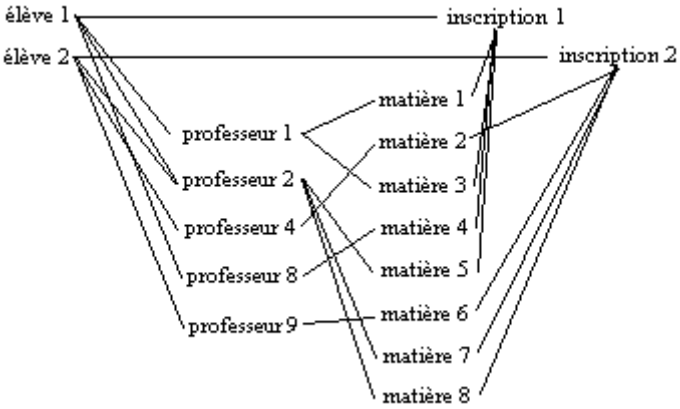
(Prenons un exemple comparatif où des élèves ont des cours donnés par des professeurs leur enseignant certaines

matières (les enseignants étant pluridisciplinaires)

- **Le modèle hiérarchique:** l'information est organisée de manière arborescente, accessible uniquement à partir de la racine de l'arbre hiérarchique. Le problème est que les points d'accès à l'information sont trop restreints.



- **Le modèle réseau:** toutes les informations peuvent être associées les unes aux autres et servir de point d'accès. Le problème est la trop grande complexité d'une telle organisation.



- **Le modèle relationnel**: toutes les relations entre les objets contenant les informations sont décrites et représentées sous la forme de tableaux à 2 dimensions.

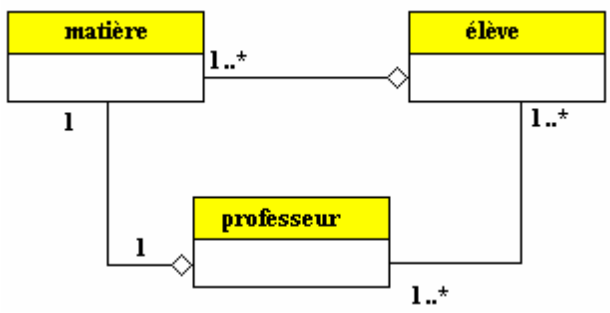
élève 1	matière 1
élève 1	matière 3
élève 1	matière 4
élève 1	matière 5
élève 2	matière 2
élève 2	matière 6
élève 2	matière 7
élève 2	matière 8

professeur 1	matière 1
professeur 1	matière 3
professeur 2	matière 5
professeur 2	matière 7
professeur 2	matière 8
professeur 4	matière 2
professeur 8	matière 4
professeur 9	matière 6

Dans ce modèle, la gestion des données (insertion, extraction,…) fonctionne selon la théorie mathématique de l'algèbre relationnelle. C'est le modèle qui allie une grande indépendance vis à vis des données à une simplicité de description.

- **Le modèle par déduction :** comme dans le modèle relationnel les données sont décrites et représentées sous la forme de tableaux à 2 dimensions. La gestion des données (insertion, extraction,…) fonctionne selon la théorie mathématique du calcul dans la logique des prédicats. Il ne semble exister de SGBD commercial directement basé sur ce concept. Mais il est possible de considérer un programme Prolog (programmation en logique) comme une base de données car il intègre une description des données. Ce sont plutôt les logiciels de réseaux sémantiques qui sont concernés par cette approche (cf. logiciel AXON).

- **Le modèle objet :** les données sont décrites comme des classes et représentées sous forme d'objets, un modèle **relationnel-objet** devrait à terme devenir le modèle de base.



L'expérience montre que le modèle relationnel s'est imposé parce qu'il était le plus simple en terme d'indépendance des données par rapport aux applications et de facilité de représenter les données dans notre esprit. C'est celui que nous décrirons succinctement dans la suite de ce chapitre.

2. Le modèle de données relationnelles

Défini par EF Codd de la société IBM dès 1970, ce modèle a été amélioré et rendu opérationnel dans les années 80 sous la forme de SBGD-R (SGBD Relationnels). Ci-dessous une liste non exhaustive de tels SGBD-R :

Access de Microsoft,
Oracle,
DB2 d'IBM,
Interbase de Borland,
SQL server de microsoft,
Informix,
Sybase,
MySQL,
PostgreSQL, ….

Nous avons déjà vu dans un précédent chapitre, la notion de relation binaire : une relation binaire R est un sous-ensemble d'un produit cartésien de deux ensembles finis E et F que nous nommerons domaines de la relation R :

$$R \subseteq E \times F$$

Cette définition est généralisable à n domaines, nous dirons que **R** est une relation n-aire sur les domaines E_1 , E_2 , … , E_n si et seulement si :

$$R \subseteq E_1 \times E_2 \ldots \times E_n$$

Les ensembles E_k peuvent être définis comme en mathématiques : en extension ou en compréhension :

E_k = { 12 , 58 , 36 , 47 } *en extension*

E_k = { x / (x est entier) **et(** x ∈ [1, 20]) } *en compréhension*

Notation

si nous avons: $R = \{ (v_1 , v_2 \ldots , v_n) \}$,
Au lieu d'écrire : $(v_1 , v_2 \ldots , v_n) \in R$, on écrira $R(v_1 , v_2 , \ldots , v_n)$

Exemple de déclarations de relations :

Passager (nom, prénom, n° de vol, nombre de bagages) , cette relation contient les informations utiles sur un passager d'une ligne aérienne.

Personne (nom, prénom) , cette relation caractérise une personne avec deux attributs

Enseignement (professeur, matière) , cette relation caractérise un enseignement avec le nom de la matière et le professeur qui l'enseigne.

Schéma d'une relation

On appelle schéma de la relation R : $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$
Où $(a_1 , a_2 \ldots , a_n)$ sont appelés les **attributs**, chaque attribut a_k indique comment est utilisé dans la relation R le domaine E_k , chaque attribut prend sa valeur dans le domaine qu'il définit, nous notons val(a_k)= v_k où v_k est un élément (une valeur) quelconque de l'ensemble E_k (domaine de l'attribut a_k).

Convention : lorsqu'il n'y a pas de valeur associée à un attribut dans un n-uplet, on convient de lui mettre une valeur spéciale notée **null**, indiquant l'absence de valeur de l'attribut dans ce n-uplet.

Degré d'une relation

On appelle degré d'une relation, le nombre d'attributs de la relation.

Exemple de schémas de relations :

Passager (nom : chaîne, prénom : chaîne, n° de vol : entier, nombre de bagages : entier) relation de degré 4.

Personne (nom : chaîne, prénom : chaîne) relation de degré 2.

Enseignement (professeur : ListeProf, matière : ListeMat) relation de degré 2.

*Attributs : prenons le schéma de la relation **Enseignement***

Enseignement (professeur : ListeProf, matière : ListeMat). C'est une relation binaire (degré 2) sur les deux domaines ListeProf et ListeMat. L'attribut professeur joue le rôle d'un paramètre formel et le domaine ListeProf celui du type du paramètre.

Supposons que :

ListeProf ={ Poincaré, Einstein, Lavoisier, Raimbault , Planck }
ListeMat = { mathématiques, poésie , chimie , physique }

L'attribut professeur peut prendre toutes valeurs de l'ensemble ListeProf :

Val(professeur) = Poincaré, …. , Val(professeur) = Raimbault

Si l'on veut dire que le poste d'enseignant de chimie n'est pas pourvu on écrira :

Le couple (**null** , chimie) est un couple de la relation **Enseignement.**

Enregistrement dans une relation

Un n-uplet $(val(a_1), val(a_2) \ldots , val(a_n)) \in R$ est appelé un enregistrement de la relation R. Un enregistrement est donc constitué de valeurs d'attributs.

Dans l'exemple précédent (Poincaré , mathématiques), (Raimbault , poésie) , (**null** , chimie) sont trois enregistrements de la relation **Enseignement**.

Clef d'une relation

Si l'on peut caractériser d'une façon **bijective** tout n-uplet d'attributs $(a_1 , a_2 \ldots , a_n)$ avec seulement un sous-ensemble restreint $(a_{k1} , a_{k2} \ldots , a_{kp})$ avec p < n , de ces attributs, alors ce sous-ensemble est appelé une **clef** de la relation. Une relation peut avoir plusieurs clefs, nous choisissons l'une d'elle en la désignant comme **clef primaire de la relation**.

Clef minimale d'une relation

On a intérêt à ce que la clef **primaire soit minimale** en nombre d'attributs, car il est clair que si un sous-ensemble à p attributs $(a_{k1}, a_{k2} \ldots, a_{kp})$ est une clef, tout sous-ensemble à p+m attributs dont les p premiers sont les $(a_{k1}, a_{k2} \ldots, a_{kp})$ est aussi une clef :
$(a_{k1}, a_{k2} \ldots, a_{kp}, a_0, a_1)$
$(a_{k1}, a_{k2} \ldots, a_{kp}, a_{10}, a_5, a_9, a_2)$ sont aussi des clefs etc…

Il n'existe aucun moyen méthodique formel général pour trouver une clef primaire d'une relation, il faut observer attentivement. Par exemple :

- Le code Insee est une clef primaire permettant d'identifier les personnes.
- Si le couple (nom, prénom) peut suffire pour identifier un élève dans une classe d'élèves, et pourrait être choisi comme clef primaire, il est insuffisant au niveau d'un lyçée où il est possible que l'on trouve plusieurs élèves portant le même nom et le même premier prénom ex: (martin, jean).

Convention : on souligne dans l'écriture d'une relation dont on a déterminé une clef primaire, les attributs faisant partie de la clef.

Clef secondaire d'une relation

Tout autre clef de la relation qu'une clef primaire (minimale) , exemple :
Si $(a_{k1}, a_{k2} \ldots, a_{kp})$ est un clef primaire de R
$(a_{k1}, a_{k2} \ldots, a_{kp}, a_0, a_1)$ et $(a_{k1}, a_{k2} \ldots, a_{kp}, a_{10}, a_5, a_9, a_2)$ sont des clefs secondaires.

Clef étrangère d'une relation

Soit $(a_{k1}, a_{k2} \ldots, a_{kp})$ un p-uplet d'attributs d'une relation R de degré n. [$R(a_1 : E_1, a_2 : E_2, \ldots, a_n : E_n)$]

Si $(a_{k1}, a_{k2} \ldots, a_{kp})$ est une clef primaire d'une autre relation Q on dira que $(a_{k1}, a_{k2} \ldots, a_{kp})$ est une clef étrangère de R.

Convention : on met un # après chaque attribut d'une clef étrangère.

Exemple de clef secondaire et clef étrangère :

Passager (nom# : chaîne, prénom# : chaîne , n° de vol : entier, nombre de bagages : entier, <u>n° client</u> : entier) relation de degré 5.

Personne (<u>nom</u> : chaîne, <u>prénom</u> : chaîne , âge : entier, civilité : Etatcivil) relation de degré 4.

<u>n° client</u> est une clef primaire de la relation **Passager**.
(nom, n° client) est une clef secondaire de la relation **Passager**.
(nom, n° client , n° de vol) est une clef secondaire de la relation **Passager**….etc

(<u>nom</u> , prénom) est une clef primaire de la relation **Personne**, comme (<u>nom</u># , prénom#) est aussi un couple d'attributs de la relation **Passager**, c'est une clef étrangère de la relation **Passager**.

On dit aussi que dans la relation **Passager**, le couple d'attributs (nom# , prénom#) réfère à la relation **Personne**.

Règle d'intégrité référentielle

Toutes les valeurs d'une clef étrangère $(V_{k1}, V_{k2} \ldots, V_{kp})$ se retrouvent comme valeur de la clef primaire de la relation référée (ensemble des valeurs de la clef étrangère est **inclus** au sens large dans l'ensemble des valeurs de la clef primaire)

Reprenons l'exemple précédent

(_nom , prénom) est une clef étrangère de la relation **Passager**, c'est donc une clef primaire de la relation **Personne** : les domaines (liste des noms et liste des prénoms associés au nom doivent être strictement les mêmes dans **Passager** et dans **Personne**, nous dirons que la clef étrangère respecte la contrainte d'intégrité référentielle.

Règle d'intégrité d'entité

Aucun des attributs participant à une clef primaire ne peut avoir la valeur **null**.

Nous définirons la valeur **null**, comme étant une valeur spéciale n'appartenant pas à un domaine spécifique mais ajoutée par convention à tous les domaines pour indiquer qu'un champ n'est pas renseigné.

Représentation sous forme tabulaire

Reprenons les relations Passager et Personne et figurons un exemple pratique de valeurs des relations.

Passager (nom# : chaîne, prénom# : chaîne, n° de vol : entier, nombre de bagages : entier, n° client : entier).

Personne (nom : chaîne, prénom : chaîne, âge : entier, civilité : Etatcivil) relation de degré 4.

Nous figurons les tables de valeurs des deux relations

Clef étrangère

Passager :

nom#	prénom#	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45622	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	**null**	858547
Lavoisier	Antoine	45644	4	235002

Clef primaire

Personne:

Clef primaire

nom	prénom	âge	Civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Nous remarquons que la compagnie aérienne attribue un numéro de client unique à chaque personne, c'est donc un bon choix pour une clef primaire.

Les deux tables (relations) ont deux colonnes qui portent les mêmes noms colonne **nom** et colonne **prénom**, ces deux colonnes forment une clef primaire de la table **Personne**, c'est donc une clef étrangère de **Passager** qui réfère **Personne.**

En outre, cette clef étrangère respecte la contrainte d'intégrité référentielle : la liste des valeurs de la clef étrangère dans **Passager** est incluse dans la liste des valeurs de la clef primaire associée dans **Personne**.

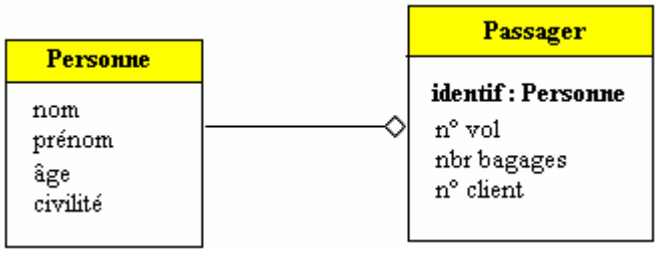


Diagramme UML modélisant la liaison Passager-Personne

Ne pas penser qu'il en est toujours ainsi, par exemple voici une autre relation **Passager2** dont la clef étrangère ne respecte pas la contrainte d'intégrité référentielle :

Passager2 :

Clef étrangère, réfère Personne

nom	prénom	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	**null**	858547
Picasso	Pablo	12896	5	458023

En effet, le couple (Picasso, Pablo) n'est pas une valeur de la clef primaire dans la table **Personne**.

Principales règles de normalisation d'une relation

1ère forme normale :
Une relation est dite en première forme normale si, chaque attribut est représenté par un identifiant unique (les valeurs ne sont pas des ensembles, des listes,…) .Ci-dessous une relation qui n'est pas en 1ère forme normale car l'attribut **n° vol** est multivalué (il peut prendre 2 valeurs) :

nom	prénom	N°vol	Nbr bagage	N° client
Einstein	Albert	~~45622 , 75906~~	2	154565
Lavoisier	Antoine	~~45644 , 45622~~	1	785154
Raimbault	Arthur	~~12896~~	2	544552
Poincaré	Henri	45644	3	781201
Picasso	Pablo	12896	5	458023

En pratique, il est très difficile de faire vérifier automatiquement cette règle, dans l'exemple proposé on pourrait imposer de passer par un masque de saisie afin que le N°vol ne comporte que 5 chiffres.

2ème forme normale :
Une relation est dite en deuxième forme normale si, elle est **en première forme normale** et si chaque attribut qui n'est pas une clef, dépend entièrement de tous les attributs qui composent la clef primaire. La relation **Personne** (nom : chaîne, prénom : chaîne , age : entier , civilité : Etatcivil) est en deuxième forme normale :

nom	prénom	âge	Civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf

Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut âge ne dépend que du nom et du prénom, de même pour l'attribut civilité.

La relation **Personne3** (nom : chaîne, prénom : chaîne , age : entier , civilité : Etatcivil) qui a pour clef primaire (nom , âge) n'est pas en deuxième forme normale :

nom	**prénom**	**âge**	**Civilité**
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut Civilité ne dépend que du nom et non pas de l'âge ! Il en est de même pour le prénom, soit il faut changer de clef primaire et prendre (nom, prénom) soit si l'on conserve la clef primaire (nom , âge) , il faut décomposer la relation **Personne3** en deux autres relations **Personne31** et **Personne32** :

nom	**âge**	**Civilité**
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Raimbault	25	célibataire
Poincaré	45	marié

Personne31(nom : chaîne, age : entier , civilité : Etatcivil) :
2ème forme normale

nom	**âge**	**prénom**
Einstein	45	Albert
Lavoisier	41	Antoine
Planck	52	Max
Raimbault	25	Arthur
Poincaré	45	Henri

Personne32(nom : chaîne, age : entier , prénom : chaîne) :
2ème forme normale

En pratique, il est aussi très difficile de faire vérifier automatiquement la mise en deuxième forme normale. Il faut trouver un jeu de données représentatif.

3ème forme normale :

Une relation est dite en troisième forme normale si chaque attribut qui ne compose pas la clef primaire, dépend directement de son identifiant et à travers une dépendance fonctionnelle. Les relations précédentes sont toutes en forme normale. Montrons un exemple de relation qui n'est pas en forme normale. Soit la relation **Personne4** (nom : chaîne, age : entier, civilité : Etatcivil, salaire : monétaire) où par exemple le salaire dépend de la clef primaire et que l'attribut civilité ne fait pas partie de la clef primaire (nom , âge) **:**

nom	**âge**	**Civilité**	**salaire**
Einstein	45	marié	1000
Lavoisier	41	marié	1000
Planck	52	veuf	1800
Raimbault	25	célibataire	1200
Poincaré	45	marié	1000

salaire = f (Civilité) =>
Pas 3ème forme normale

L'attribut salaire dépend de l'attribut civilité, ce que nous écrivons salaire = f(civilité), mais l'attribut civilité ne fait pas partie de la clef primaire clef = (nom , âge), donc **Personne4** n'est pas en 3ème forme normale **:**

Il faut alors décomposer la relation **Personne4** en deux relations **Personne41** et **Personne42** chacune en troisième forme normale:

Personne41 :

nom	âge	Civilité#
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Raimbault	25	célibataire
Poincaré	45	marié

Personne41 en 3ème forme normale, civilité clef étrangère

Personne42 :

Civilité	salaire
marié	1000
veuf	1800
célibataire	1200

Personne42 en 3ème forme normale, civilité clef primaire

En pratique, il est également très difficile de faire contrôler automatiquement la mise en troisième forme normale.

Remarques pratiques importantes pour le débutant :

- Les spécialistes connaissent deux autres formes normales. Dans ce cas le lecteur intéressé par l'approfondissement du sujet, trouvera dans la littérature, de solides références sur la question.
- Si la clef primaire d'une relation n'est composée que d'un seul attribut (choix conseillé lorsque cela est possible, d'ailleurs on trouve souvent des clefs primaires sous forme de numéro d'identification client, Insee,…) automatiquement, la relation est en 2ème forme normale, car chaque autre attribut non clef étrangère, ne dépend alors que de la valeur unique de la clef primaire.
- Penser dès qu'un attribut est fonctionnellement dépendant d'un autre attribut qui n'est pas la clef elle-même à décomposer la relation (créer deux nouvelles tables).
- En l'absence d'outil spécialisé, il faut de la pratique et être très systématique pour contrôler la normalisation.

Base de données relationnelles BD-R:

Ce sont des données structurées à travers :

- Une famille de domaines de valeurs,
- Une famille de relations n-aires,
- Les contraintes d'intégrité sont respectées par toute clef étrangère et par toute clef primaire.
- Les relations sont en 3ème forme normale. (à minima en 2ème forme normale)

Les données sont accédées et manipulées grâce à un langage appelé **langage d'interrogation** ou

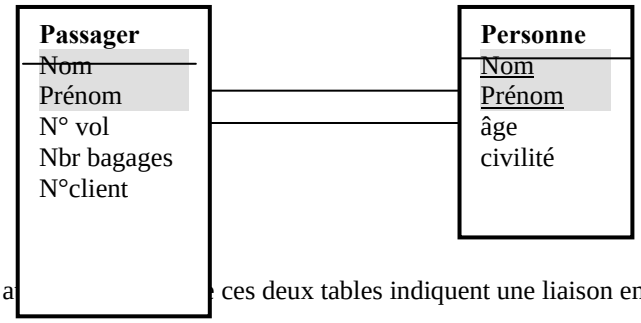
langage relationnel ou **langage de requêtes**

Système de Gestion de Base de Données relationnel :

C'est une famille de logiciels comprenant :

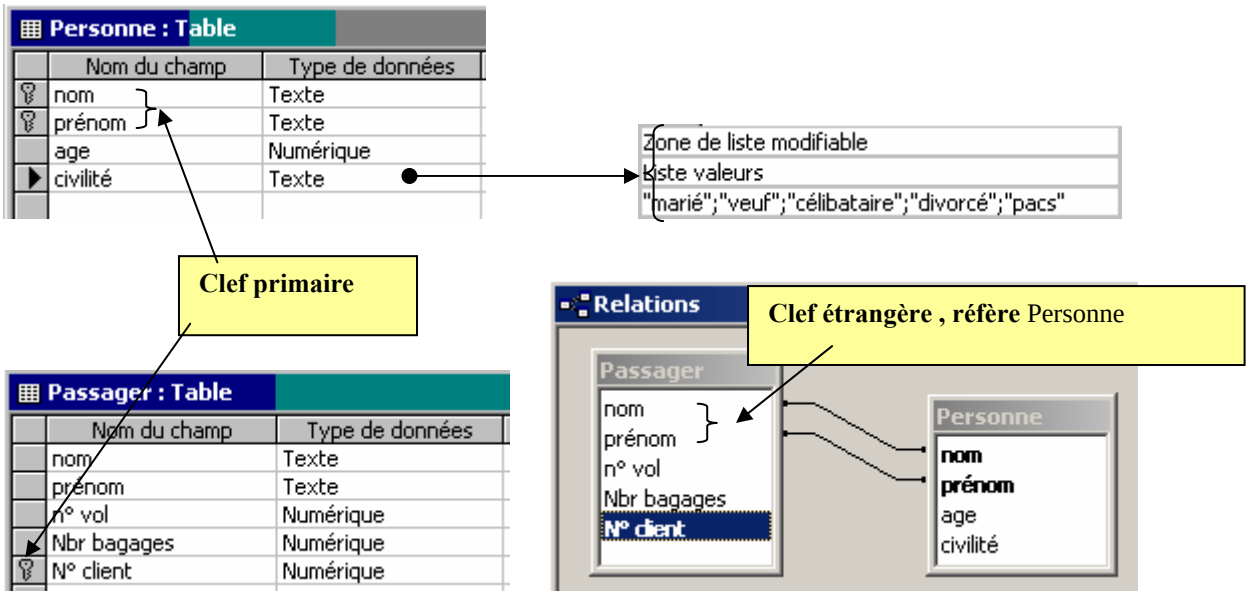
- Une BD-R.
- Un langage d'interrogation.
- Une gestion en interne des fichiers contenant les données et de l'ordonnancement de ces données.
- Une gestion de l'interface de communication avec les utilisateurs.
- La gestion de la sécurité des accès aux informations contenues dans la BD-R.

Le schéma relation d'une relation dans une BD relationnelle est noté graphiquement comme ci-dessous :



Les a ces deux tables indiquent une liaison entre les deux relations.

Voici dans le **SGBD-R Access**, la représentation des schémas de relation ainsi que la liaison sans intégrité des deux relations précédentes **Passager** et **Personne** :



Access et la représentation des enregistrements de chaque table :

Passager : Table

nom	prénom	n° vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	4	235002
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Lavoisier	Antoine	45644	1	785154
Einstein	Albert	75906	0	858547

Les enregistrements de la relation **Passager**

Personne : Table

nom	prénom	age	civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Poincaré	Henri	45	marié
Raimbault	Arthur	25	célibataire

Les enregistrements de la relation **Personne**

Les besoins d'un utilisateur d'une base de données sont classiquement ceux que l'on trouve dans tout ensemble de données structurées : insertion, suppression, modification, recherche avec ou sans critère de sélection. Dans une BD-R, ces besoins sont exprimés à travers un langage d'interrogation. Historiquement deux classes de langages relationnels équivalentes en puissance d'utilisation et de fonctionnement ont été inventées : les langages **algébriques** et les langages des **prédicats**.

Un langage relationnel n'est pas un langage de programmation : il ne possède pas les structures de contrôle de base d'un langage de programmation (condition, itération, …). **Très souvent il doit être utilisé comme complément à l'intérieur de programmes Delphi, Java, C#, ...**

Les langages d'interrogation prédicatifs sont des langages fondés sur la logique des prédicats du 1er ordre, le plus ancien s'appelle **Q**uery **B**y **E**xample QBE.

Ce sont les langages algébriques qui sont de loin les plus utilisés dans les SGBD-R du commerce, le plus connu et le plus utilisé dans le monde se dénomme le **S**tructured **Q**uery **L**anguage ou SQL. Un tel langage n'est qu'une implémentation en anglais d'opérations définies dans une algèbre relationnelle servant de modèle mathématique à tous les langages relationnels.

3. Principes fondamentaux d'une l'algèbre relationnelle

Une algèbre relationnelle est une famille d'opérateurs binaires ou unaires dont les opérandes sont des **relations**. Nous avons vu que l'on pouvait faire l'union, l'intersection, le produit cartésien de relations binaires dans un chapitre précédent, comme les relations n-aires sont des ensembles, il est possible de définir sur elle une algèbre opératoire utilisant les opérateurs classiques ensemblistes, à laquelle on ajoute quelques opérateurs spécifiques à la manipulation des données.

Remarque pratique :

> La phrase "**tous les n-uples sont distincts, puisqu'éléments d'un même ensemble nommé relation**" se transpose en pratique en la phrase **" toutes les lignes d'une même table nommée relation, sont distinctes** (même en l'absence de clef primaire explicite)".

Nous exhibons les opérateurs principaux d'une algèbre relationnelle et nous montrerons pour chaque opération, un exemple sous forme.

Union de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R ∪ Q de même degré et de même domaine contenant les enregistrements différents des deux relations R et Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R ∪ Q

Q :

nom	âge	Civilité
Lupin	45	célibataire
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Lupin	45	célibataire
Mozart	34	veuf
Gandhi	64	célibataire

Remarque : (Planck, 52, veuf) ne figure qu'une seule fois dans la table R ∪ Q.

Intersection de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R ∩ Q de même degré et de même domaine contenant les enregistrements communs aux deux relations R et Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R ∩ Q

Q :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf

Différence de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R – Q de même degré et de même domaine contenant les enregistrements qui sont présents dans R mais qui ne sont pas dans Q (on exclut de R les enregistrements qui appartiennent à R ∩ Q) :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R – Q

Q :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Lavoisier	41	marié

Produit cartésien de 2 relations

Soient R et Q deux relations de domaine et de degré quelconques (degré(R)=n, degré(Q)=p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

On peut calculer la nouvelle relation S = R x Q de degré n + p et de domaine égal à l'union des domaines de R et de Q contenant tous les couples d'enregistrements à partir d'enregistrements présents dans R et d'enregistrements présents dans Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R x Q

Q :

ville	km
Paris	874
Rome	920

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Einstein	45	marié	Rome	920
Lavoisier	41	marié	Paris	874
Lavoisier	41	marié	Rome	920
Planck	52	Veuf	Paris	874
Planck	52	Veuf	Rome	920

Selection ou Restriction d'une relation

Soit R une relation, soit $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$ le schéma de cette relation.

Soit Cond(a_1, a_2 , … , a_n) une expression booléenne classique (expression construite sur les attributs avec les connecteurs de l'algèbre de Boole et les opérateurs de comparaison < , > , = , >= , <= , <>)

On note S = **select** (Cond(a_1, a_2 , … , a_n) , R), la nouvelle relation S construite ayant le même schéma que R soit $S(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$, qui ne contient que les enregistrements de R qui satisfont à la condition booléenne Cond(a_1, a_2 , … , a_n).

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

Cond(a_1, a_2 , … , a_n) = { âge > 42 **et** ville=Paris }

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Gandhi	64	célibataire	Paris	258

Select ({ âge > 42 et ville=Paris}, R) signifie que l'on ne recopie dans S que les enregistrements de R constitués des personnes ayant séjourné à Paris et plus âgées que 42 ans.

Projection d'une relation

Soit R une relation, soit $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$ le schéma de cette relation.
On appelle S = **proj**$(a_{k1} , a_{k2} \ldots , a_{kp})$ la projection de R sur un sous-ensemble restreint $(a_{k1} , a_{k2} \ldots , a_{kp})$ avec p < n , de ses attributs, la relation S ayant pour schéma le sous-ensemble des attributs $S(a_{k1} : E_{k1}, a_{k2} : E_{k2} ,\ldots, a_{kp} : E_{kp})$ et contenant les enregistrements différents obtenus en ne considérant que les attributs $(a_{k1} , a_{k2} \ldots , a_{kp})$.

Exemple

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

S1 = **proj**(nom , civilité)

nom	Civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Gandhi	célibataire
Lavoisier	marié
Planck	Veuf

S2 = **proj**(nom , âge)

nom	âge
Einstein	45
Mozart	32
Lupin	42
Gandhi	64
Lavoisier	41
Planck	52

S3 = **proj**(nom , ville)

nom	ville
Einstein	Paris
Mozart	Rome
Lupin	Paris
Einstein	Venise
Gandhi	Paris
Lavoisier	Rome
Planck	Rome

S4 = **proj**(ville)

ville
Paris
Rome
Venise

Que s'est-il passé pour Mr Einstein dans S2 ?

Einstein	45	marié	Paris	874
Einstein	45	marié	Venise	981
Einstein		**marié**		

Lors de la recopie des enregistrements de R dans S2 on a ignoré les attributs âge, ville et km, le couple (Einstein, marié) ne doit se retrouver qu'une seule fois car une relation est un ensemble et ses éléments sont tous distincts.

Jointure de deux relations

Soit Soient R et Q deux relations de domaine et de degré quelconques (degré(R) = n, degré(Q) = p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

soit R x Q leur produit cartésien de degré n + p et de domaine D union des domaines de R et de Q.
Soit un ensemble $(a_1, a_2 , \ldots , a_{n+p})$ d'attributs du domaine D de R x Q.

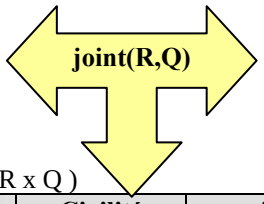
> La relation **joint**(R,Q) = **select** (Cond(a_1, a_2 , … , a_{n+p}) , R x Q), est appelée jointure de R et de Q (c'est donc une sélection de certains attributs sur le produit cartésien).

Une jointure couramment utilisée en pratique, est celle qui consiste en la sélection selon une condition d'égalité entre deux attributs, les personnes de "l'art relationnel" la dénomment alors l'**équi-jointure**.

Exemples de 2 jointures :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf



Q :

ville	km
Paris	874
Rome	920

1°) S = **select** (km < 900 , R x Q)

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Lavoisier	41	marié	Paris	874
Planck	52	veuf	Paris	874

2°) S = **select** (km < 900 **et** Civilité = veuf, R x Q)

nom	âge	Civilité	ville	km
Planck	52	veuf	Paris	874

Nous nous plaçons maintenant du point de vue pratique, non pas de l'administrateur de BD mais de l'utilisateur uniquement concerné par l'extraction des informations contenues dans une BD-R.

Un SGBD permet de gérer une base de données. A ce titre, il offre de nombreuses fonctionnalités supplémentaires à la gestion d'accès simultanés à la base et à un simple interfaçage entre le modèle logique et le modèle physique : il sécurise les données (en cas de coupure de courant ou autre défaillance matérielle), il permet d'accéder aux données de manière confidentielle (en assurant que seuls certains utilisateurs ayant des mots de passe appropriés, peuvent accéder à certaines données), il ne permet de mémoriser des données que si elles sont du type abstrait demandé : on dit qu'il vérifie leur intégrité (des données alphabétiques ne doivent pas être enregistrées dans des emplacements pour des données numériques,...)

Actuellement, une base de données n'a pas de raison d'être sans son SGBD. Aussi, on ne manipule que des bases de données correspondant aux SGBD qui les gèrent : il vous appartient de choisir le SGBD-R qui vous convient (il faut l'acheter auprès de vendeurs qui généralement, vous le fournissent avec une application de manipulation visuelle, ou bien utiliser les SGBD-R qui vous sont livrés gratuitement avec certains environnements de développement comme Borland studio ou Visual Studio ou encore utiliser les produits gratuits comme mySql).

Lorsque l'on parle d'utilisateur, nous entendons l'application utilisateur, car l'utilisateur final n'a pas besoin de connaître quoique ce soit à l'algèbre relationnelle, il suffit que l'application utilisateur communique avec lui et interagisse avec le SGBD.

Une application doit pouvoir "parler" au SGBD : elle le fait par le moyen d'un langage de manipulation des données. Nous avons déjà précisé que la majorité des SGBD-R utilisait un langage relationnel ou de requêtes nommé SQL pour manipuler les données.

4. SQL et Algèbre relationnelle

Requête

Les requêtes sont des questions posées au SGBD, concernant une recherche de données contenues dans une ou plusieurs tables de la base.

Par exemple, on peut disposer d'une table définissant des clients (noms, prénoms, adresses, n°de client) et d'une autre table associant des numéros de clients avec des numéros de commande d'articles, et vouloir poser la question suivante : quels sont les noms des clients ayant passé des commandes ?

Une requête est en fait, une instruction de type langage de programmation, respectant la norme SQL, permettant de réaliser un tel questionnement. L'exécution d'une requête permet d'extraire des données en provenance de tables de la base de données : ces données réalisent ce que l'on appelle une **projection** de champs (en provenance de plusieurs tables). Le résultat d'exécution d'une requête est une table constituée par les réponses à la requête.

Le SQL permet à l'aide d'instructions spécifiques de manipuler des données présentes à l'intérieur des tables :

Instruction SQL	Actions dans la (les) table(s)
INSERT INTO <…>	Ajout de lignes
DELETE FROM <…>	Suppression de lignes
TRUNCATE TABLE <…>	Suppression de lignes
UPDATE <…>	Modification de lignes
SELECT <…>**FROM** <…>	Extraction de données

Ajout, suppression et modification sont les trois opérations typiques de la **mise à jour** d'une BD. L'extraction concerne la **consultation** de la BD.

Il existe de nombreuses autres instructions de création, de modification, de suppression de tables, de création de clefs, de contraintes d'intégrités référentielles, création d'index, etc… Nous nous attacherons à donner la traduction en SQL des opérateurs principaux de l'algèbre relationnelle que nous venons de citer.

Traduction en SQL des opérateurs relationnels

C'est l'instruction SQL "**SELECT** <…>**FROM** <…>" qui implante tous ces opérateurs. Tous les exemples utiliseront la relation R = TableComplete suivante et l'interpréteur SQL d'**Access** :

TableComplete : Table

nom	âge	civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

La relation initiale : R=TableComplete

Projection d'une relation R

S = **proj**(a_{k1} , a_{k2} … , a_{kp})

SQL : **SELECT DISTINCT** a_{k1} , a_{k2} … , a_{kp} **FROM** R

Instruction SQL version opérateur algèbre :
SELECT DISTINCT nom , civilité **FROM** Tablecomplete

Lancement de la requête SQL :

RequêteS1 : Requête Sélection

```
SELECT DISTINCT [nom], [civilité]
FROM TableComplete;
```

Le mot DISTINCT assure que l'on obtient bien une nouvelle relation.

Table obtenue après requête :

RequêteS1 : Requête Sélection

nom	civilité
Einstein	marié
Gandhi	célibataire
Lavoisier	marié
Lupin	Veuf
Mozart	marié
Planck	Veuf

Instruction SQL non relationnelle (tout même redondant) :
SELECT nom , civilité **FROM** Tablecomplete

Lancement de la requête SQL :

RequêteS1 : Requête Sélection

```
SELECT [nom], [civilité]
FROM TableComplete;
```

Table obtenue après requête :

RequêteS1 : Requête Sélé

nom	civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Einstein	marié
Gandhi	célibataire
Lavoisier	marié
Lupin	Veuf
Planck	Veuf

Remarquons dans le dernier cas **SELECT** nom , civilité **FROM** Tablecomplete que la table obtenue n'est qu'une extraction de données, mais qu'en aucun cas elle ne constitue une relation puisqu'une relation est un ensemble et que les enregistrements sont tous distincts!

Une autre projection sur la même table :

Instruction SQL version opérateur algèbre :
SELECT DISTINCT nom , ville **FROM** Tablecomplete

Lancement de la requête SQL :

RequêteS2 : Requête Sélection

```
SELECT DISTINCT [nom], [ville]
FROM TableComplete;
```

Table obtenue après requête :

RequêteS2 : Requête Sélé

nom	ville
Einstein	Paris
Einstein	Venise
Gandhi	Paris
Lavoisier	Rome
Lupin	Paris
Mozart	Rome
Planck	Rome

Sélection-Restriction

S = **select** (Cond(a_1, a_2 , … , a_n) , R)

SQL : **SELECT** * **FROM** R **WHERE** Cond(a_1, a_2 , … , a_n)

*Le symbole * signifie toutes les colonnes de la table (tous les attributs du schéma)*

Instruction SQL version opérateur algèbre :
SELECT * FROM Tablecomplete **WHERE** âge > 42 **AND** ville = Paris

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection

```
SELECT * FROM TableComplete
WHERE [âge]>42 AND [ville]="Paris";
```

Table obtenue après requête :

TableComplete Requête : Requête Sélection

nom	âge	civilité	ville	km
Einstein	45	marié	Paris	874
Gandhi	64	célibataire	Paris	258

On a sélectionné toutes les personnes de plus de 42 ans ayant séjourné à Paris.

Combinaison d'opérateur projection-restriction

Projection distincte et sélection :
SELECT DISTINCT nom , civilité, âge **FROM** Tablecomplete **WHERE** âge >= 45

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection

```
SELECT DISTINCT [nom], [civilité], [âge]
FROM TableComplete
WHERE [âge]>=45 ;
```

Table obtenue après requête :

TableComplete Requête : Requête Sé

nom	civilité	âge
Einstein	marié	45
Gandhi	célibataire	64
Planck	Veuf	52

On a sélectionné toutes les personnes d'au moins 45 ans et l'on ne conserve que leur nom, leur civilité et leur âge.

Intersection, union, différence,

S = **R** ∩ **Q**

SQL : **SELECT** * **FROM** R **INTERSECT SELECT** * **FROM** Q

S = **R** ∪ **Q**

SQL : **SELECT** * **FROM** R **UNION SELECT** * **FROM** Q

S = **R** – **Q**

SQL : **SELECT** * **FROM** R **MINUS SELECT** * **FROM** Q

Produit cartésien

S = **R x Q**

SQL : **SELECT** * **FROM** R , Q

Afin de ne pas présenter un exemple de table produit trop volumineuse, nous prendrons comme opérandes du produit cartésien, deux tables contenant peu d'enregistrements :

Personne : Table

nom	prénom	age	civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié

Employeur : Table

nom	n°Insee
Université	12112472
Collège	25478550
Institut	54578559

Produit cartésien :
SELECT * FROM Employeur , Personne

Employeur Requête : Requête Sélection

Personne.nom	prénom	age	civilité	Employeur.nom	n°Insee
Einstein	Albert	45	marié	Université	12112472
Lavoisier	Antoine	41	marié	Université	12112472
Einstein	Albert	45	marié	Collège	25478550
Lavoisier	Antoine	41	marié	Collège	25478550
Einstein	Albert	45	marié	Institut	54578559
Lavoisier	Antoine	41	marié	Institut	54578559

Nous remarquons qu'en apparence l'attribut **nom** se retrouve dans le domaine des deux relations ce qui semble contradictoire avec l'hypothèse "Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs). En fait ce n'est pas un attribut commun puisque les valeurs sont différentes, il s'agit plutôt de deux attributs différents qui ont la même identification. Il suffit de préfixer l'identificateur par le nom de la relation (Personne.nom et Employeur.nom).

Combinaison d'opérateurs : projection - produit cartésien

SELECT Personne.nom , prénom, civilité, n°Insee **FROM** Employeur , Personne

On extrait de la table produit cartésien uniquement 4 colonnes :

Employeur Requête : Requête Sélection

nom	prénom	civilité	n°Insee
Einstein	Albert	marié	12112472
Lavoisier	Antoine	marié	12112472
Einstein	Albert	marié	25478550
Lavoisier	Antoine	marié	25478550
Einstein	Albert	marié	54578559
Lavoisier	Antoine	marié	54578559

Jointure de deux relations

Soient R et Q deux relations de domaine et de degré quelconques (degré(R) = n, degré(Q) = p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

La jointure **joint**(R,Q) = **select** (Cond(a_1, a_2 , … , a_{n+p}) , R x Q).

SQL : **SELECT** * **FROM** R ,Q **WHERE** Cond(a_1, a_2 , … , a_{n+p})

Employeur : Table

nom	n°Insee
Université	12112472
Collège	25478550
Institut	54578559

Passager : Table

nom	prénom	n° vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	4	235002
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Lavoisier	Antoine	45644	1	785154
Einstein	Albert	75906	0	858547
		0	0	0

Employeur RequêteJoin : Requête Sélection

```
SELECT *
FROM Employeur, Passager
WHERE Passager.[n° vol] > 20000;
```

Employeur RequêteJoin : Requête Sélection

Employeur.no	n°Insee	Passager.	prénom	n° vol	Nbr bagages	N° client
Université	12112472	Einstein	Albert	45622	2	154565
Université	12112472	Lavoisier	Antoine	45644	1	785154
Université	12112472	Poincaré	Henri	45644	3	781201
Université	12112472	Einstein	Albert	75906	0	858547
Université	12112472	Lavoisier	Antoine	45644	4	235002
Collège	25478550	Einstein	Albert	45622	2	154565
Collège	25478550	Lavoisier	Antoine	45644	1	785154
Collège	25478550	Poincaré	Henri	45644	3	781201
Collège	25478550	Einstein	Albert	75906	0	858547
Collège	25478550	Lavoisier	Antoine	45644	4	235002
Institut	54578559	Einstein	Albert	45622	2	154565
Institut	54578559	Lavoisier	Antoine	45644	1	785154
Institut	54578559	Poincaré	Henri	45644	3	781201
Institut	54578559	Einstein	Albert	75906	0	858547
Institut	54578559	Lavoisier	Antoine	45644	4	235002

Remarque pratique importante

Le langage SQL est plus riche en fonctionnalités que l'algèbre relationnelle. En effet SQL intègre des possibilités de calcul (numériques et de dates en particulier).

Soit une table de tarifs de produit avec des prix hors taxe:

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

1°) Usage d'un opérateur multiplicatif : calcul de la nouvelle table des tarifs TTC abondés de la TVA à 20% sur le prix hors taxe.

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Tarifs TTc

Tarifs TTc : Requête Sélection

```
SELECT Article , PrixHT*1.20 AS PrixTTC
FROM Tarifs;
```

Tarifs TTc : Requête Sélection

Article	PrixTTC
chaise	12
table	120
Téléviseur	1200

2°) Usage de la fonction intégrée SUM : calcul du total des prix HT.

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Total HT

Total HT : Requête Sélection

```
SELECT SUM(PrixHT) AS Total
FROM Tarifs;
```

Total HT : Requête Sélection

Total
1 110,00 €

ADO .Net : données relationnelles de .Net 2.0

Plan général:

1. Le modèle DataSet dans ADO.Net 2.0

DataSet
DataTable
DataColumn
DataRow
DataRelation

2. Création et utilisation d'un DataSet

Création d'un DataSet
Création de 2 DataTable
Définition du schéma de colonnes de chaque table par des DataColumn.
Définition d'une clef primaire.
Ajout des 2 tables au DataSet.
Ajout d'une relation entre les deux tables.
Exemple de création de données dans le DataSet
Affichage des données d'un DataSet
Sauvegarde des données du DataSet aux formats XML et XSL

Introduction

Nous avons vu dans un chapitre précédent que les fichiers simples peuvent être accédés par des flux, dès que l'organisation de l'information dans le fichier est fortement structurée les flux ne sont plus assez puissants pour fournir un accès souple aux données en particulier pour des applications à architecture multi-tiers (client/serveur, internet,…) et spécialement aux bases de données.

ADO .NET est un regroupement de types (classes, interfaces, …) dans l'espace de nom **System.Data** construits par Microsoft afin de manipuler des données structurées dans le .NET Framework.

Le modèle ADO .NET du .NET Framework fournit au développeur un ensemble d'éléments lui permettant de travailler sur des données aussi bien en mode connecté qu'en mode déconnecté (ce dernier mode est le mode préférentiel d' ADO .NET car c'est celui qui est le plus adapté aux architectures multi-tiers). ADO .NET est indépendant du mode de stockage des données : les classes s'adaptent automatiquement à l'organisation

ADO .NET permet de traiter des données situées dans des bases de données selon le modèle relationnel mais il supporte aussi les données organisées selon le modèle hiérarchique.

ADO .NET échange toutes ses informations au format XML

L'entité la plus importante d' ADO .NET permettant de gérer les données en local dans une mémoire cache complétement déconnectée de la source de données (donc indépendante de cette source) est le DataSet en fait la classe **System.Data.DataSet** et la collection de classes qui lui sont liées.

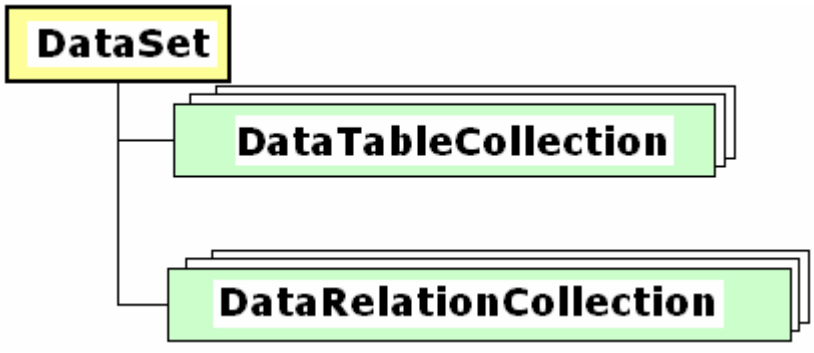
1. Le modèle DataSet dans ADO.Net 2.0

Le principe de base du DataSet est de se connecter à une source de données (par exemple une base de données) de charger toutes ses tables avec leur relations, puis ensuite de travailler en mode déconnecté sur ces tables en mémoire et enfin se reconnecter pour effectuer la mise à jour éventuelle des données.

Les classes mises en jeu lors d'une telle opération sur les données sont les suivantes :

System.Data.DataSet	
System.Data.DataTable	**System.Data.DataTableCollection**
System.Data.DataColumn	**System.Data.DataColumnCollection**
System.Data.DataRow	**System.Data.DataRowCollection**
System.Data.DataRelation	**System.Data.DataRelationCollection**
System.Data.DataConstraint	**System.Data.ConstraintCollection**
System.Data.DataView	

Un **DataSet** est en fait une collection de tables représentée par un objet de collection de la classe **DataTableCollection** et une collection de relations entres ces tables tables représentée par un objet de collection de la classe **DataRelationCollection**.



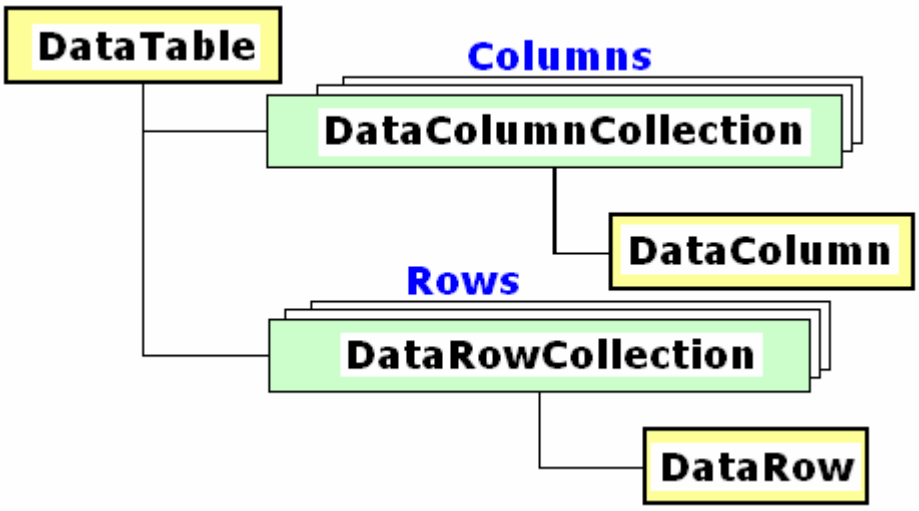
Une **DataTableCollection** est une collection (famille) d'une ou plusieurs DataTable qui représentent chacunes une table dans le DataSet.

Une table générale est une entité composée de :

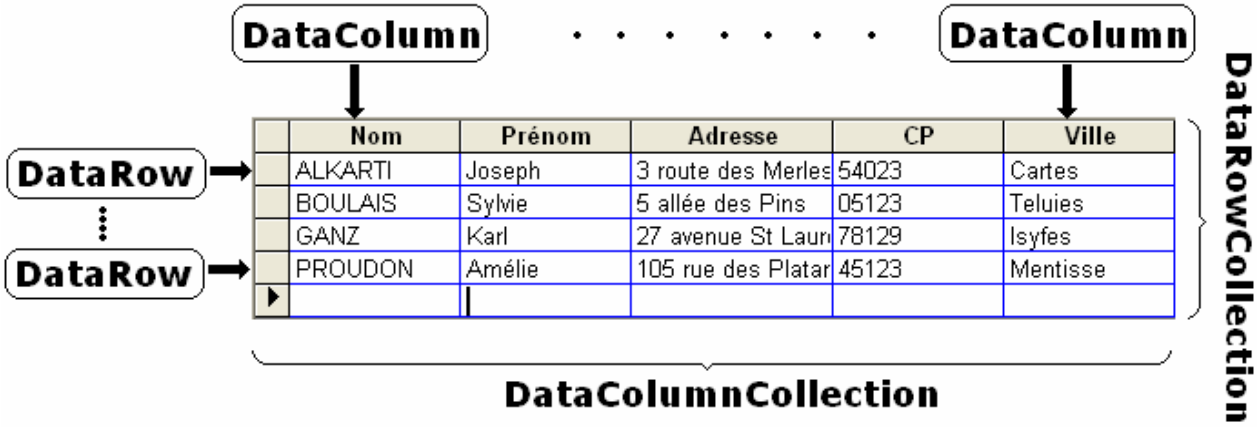
- Colonnes
- Lignes

Un objet de classe **DataTable** est composé en particulier des objets suivants :

- Une propriété **Columns** = Collection de colonnes (collection d'objets **DataColumn**)
- Une propriété **Rows** = Collection de lignes (collection d'objets **DataRow**)

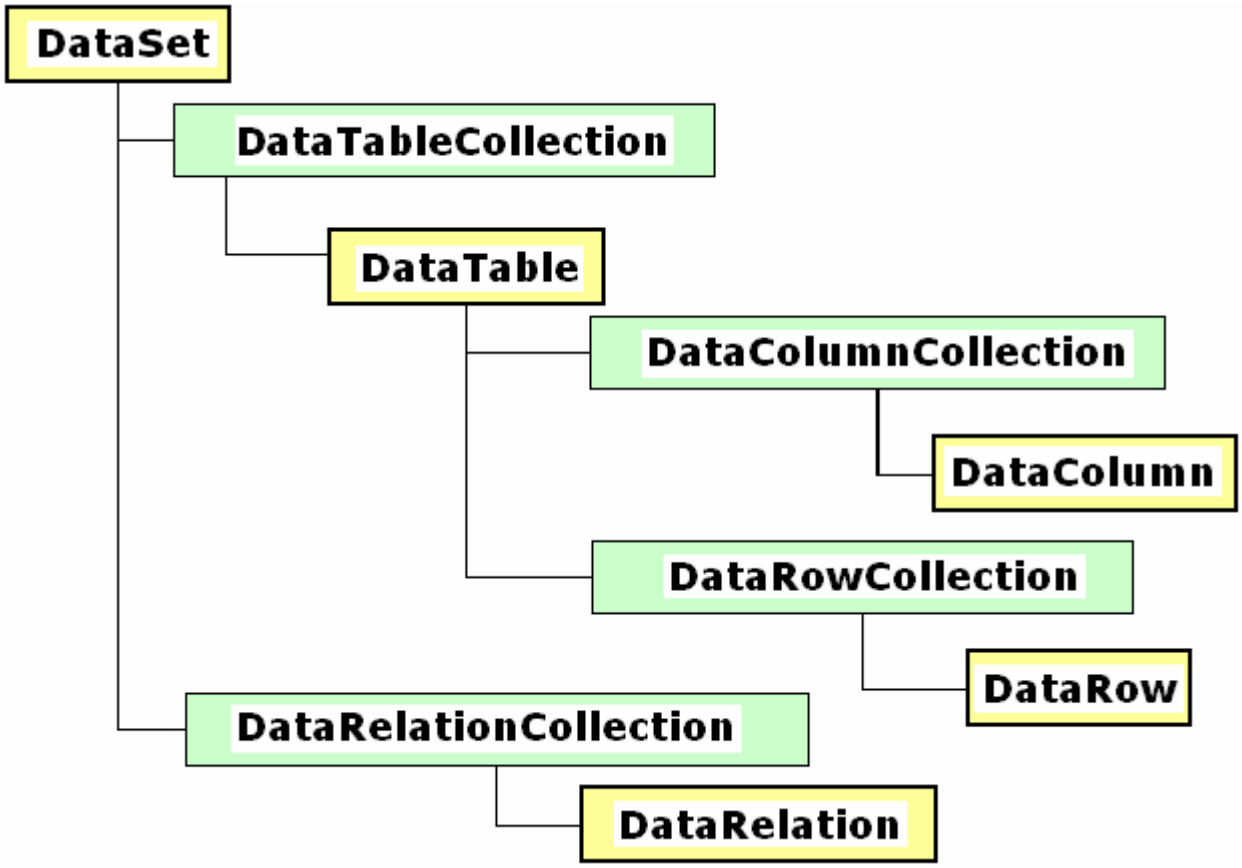


Le schéma général de la table d'un objet **DataTable** est donné par la famille des colonnes (l'objet Columns) chaque objet **DataColumn** de la collection représente une colonne de la table :

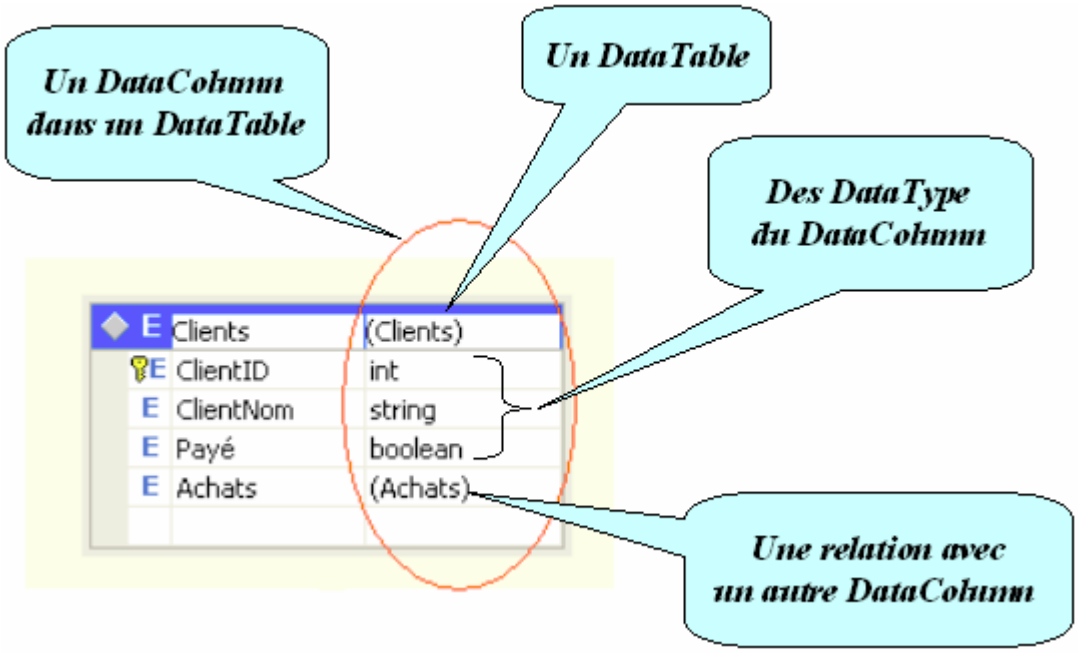


Pour résumer, un **DataSet** contient pour l'essentiel deux types d'objets :

1°) des objets **DataTable** inclus dans une **DataTableCollection**
2°) des objets **DataRelation** inclus dans une **DataRelationCollection**

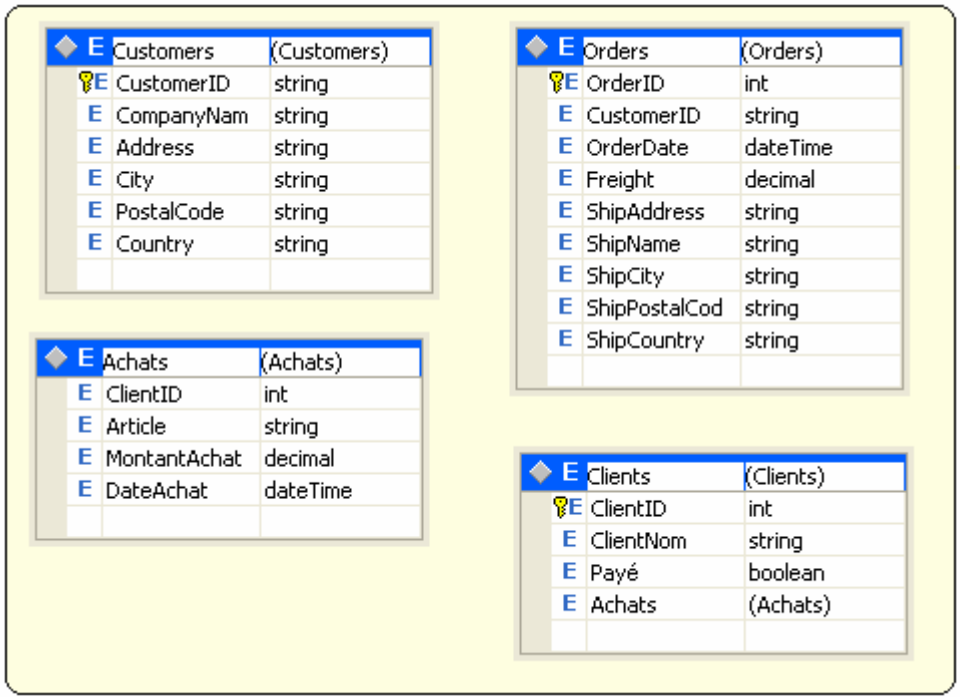


Une **DataRelationCollection** est une collection d'une ou plusieurs **DataRelation** qui représentent chacunes une relation entre deux DataTable dans le DataSet.

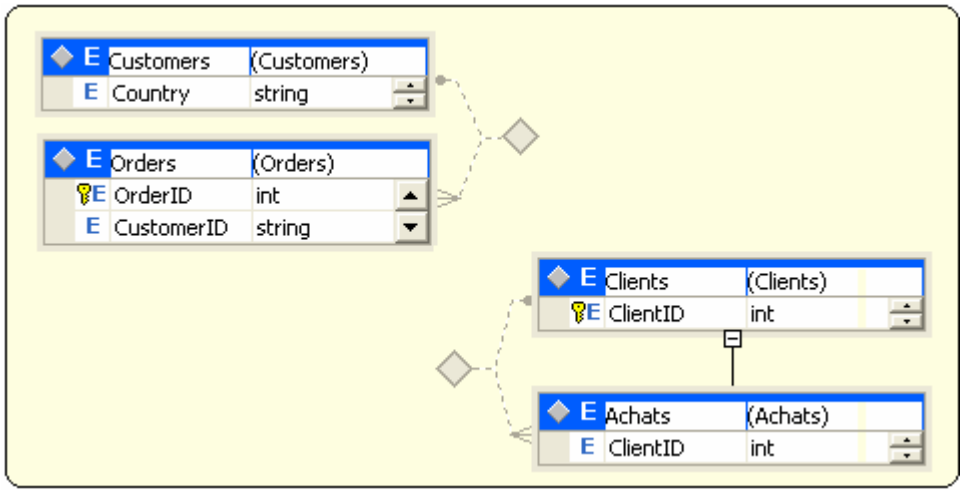


Voici une représentation fictive mais figurative d'un **DataSet** contenant 4 **DataTable** et 2 **DataRelation** :

La **DataTableCollection** du DataSet :



La **DataRelationCollection** du DataSet :



1. Création et utilisation d'un DataSet

Nous passons maintenant à la pratique d'utilisation d'un DataSet dans un programme C# de création de données selon la démarche suivante :

Phase de mise en place :

- Création d'un DataSet (un magasin).
- Création de 2 DataTable (une table client, une table achats).
- Définition du schéma de colonnes de chaque table par des DataColumn.
- Définition d'une clef primaire.
- Ajout des 2 tables au DataSet.
- Ajout d'une relation entre les deux tables.

Phase de manipulation des données proprement dites :

- Ajouter, supprimer ou modifier des données ligne par ligne dans chaque table.

Création d'un DataSet (un magasin) :

```
private DataSet unDataSet = new DataSet( );
unDataSet.DataSetName = "Magasin" ;
```

Création des deux tables :

```
private DataTable tabClients = new DataTable( "Clients" );
private DataTable tabAchats = new DataTable( "Achats" );
```

Définition du schéma de 3 colonnes de la table Clients :

```
DataColumn colClientID = new DataColumn( "ClientID", typeof(int) );
DataColumn colClientNom = new DataColumn( "ClientNom" );
DataColumn colPayed = new DataColumn( "Payé", typeof(bool) );
tabClients.Columns.Add(colClientID);
tabClients.Columns.Add(colClientNom);
tabClients.Columns.Add(colPayed);
```

Définition du schéma de 4 colonnes de la table Achats :

```
DataColumn colID = new DataColumn("ClientID", typeof(int));
DataColumn colArticle = new DataColumn("Article",typeof(string));
DataColumn colDateAchat = new DataColumn( "DateAchat",typeof(DateTime));
DataColumn colMontantAchat = new DataColumn("MontantAchat", typeof(decimal));
tabAchats.Columns.Add(colID);
tabAchats.Columns.Add(colArticle);
tabAchats.Columns.Add(colMontantAchat);
tabAchats.Columns.Add(colDateAchat);
```

Définition d'une clef primaire composée de 2 colonnes :

```
tabClients.PrimaryKey = new DataColumn[ ] {colClientID, colClientNom};
```

Ajouter les 2 tables au DataSet :

```
unDataSet.Tables.Add(tabClients);
unDataSet.Tables.Add(tabAchats);
```

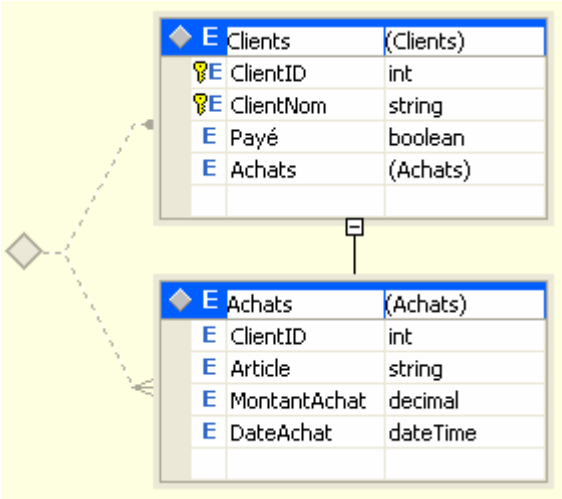
Ajouter une relation "Liste des achats" entre les 2 tables :

```
// Créer une DataRelation entre colClientID et colID
DataRelation dr = new DataRelation("Liste des achats", colClientID , colID);

// ajouter cette relation dans le DataSet
unDataSet.Relations.Add(dr);

// imbrication de relation écrite dans le fichier xml:
dr.Nested=true;
```

La validation de toutes les modifications apportées au DataSet depuis son chargement s'effectue grâce à la méthode **AcceptChanges()**, on termine donc le code par la ligne :

```
unDataSet.AcceptChanges();
```

Le DataSet ainsi construit peut maintenant travailler en lecture et en éciture sur une structure de données construite sous forme de deux tables Clients et Achats reliées entre elles comme figuré ci-desssous :



Exemple de création de données dans le DataSet :

```
enum typArticle {vélo, télévision, radio, cuisinière }

//-- remplissage direct des tables
// Déclaration d'un référence de ligne de données
DataRow newRow1;
// Création de 6 lignes de clients dans la table "Clients"
for(int i = 1; i <= 6; i++) {
  // Création de la ligne newRow1 possédant le même schéma que tabClients
  newRow1 = tabClients.NewRow();
  newRow1["ClientID"] = 1000+i;
```

```
  // Ajouter la ligne newRow1 à la table tabClients
  tabClients.Rows.Add(newRow1);
}
// Remplissage de la colonne "ClientNom" de chaque ligne de clients créée
tabClients.Rows[0]["ClientNom"] = "Legrand";
tabClients.Rows[1]["ClientNom"] = "Fischer";
tabClients.Rows[2]["ClientNom"] = "Dupont";
tabClients.Rows[3]["ClientNom"] = "Durand";
tabClients.Rows[4]["ClientNom"] = "Lamiel";
tabClients.Rows[5]["ClientNom"] = "Renoux";

// Remplissage de la colonne "ClientNom" de chaque ligne de clients créée
tabClients.Rows[0]["Payé"] = true;
tabClients.Rows[1]["Payé"] = true;
tabClients.Rows[2]["Payé"] = false;
tabClients.Rows[3]["Payé"] = false;
tabClients.Rows[4]["Payé"] = true;
tabClients.Rows[5]["Payé"] = false;

// Déclaration d'un référence de ligne de données
DataRow newRow2;

/* pour chacun des 6 clients remplissage aléatoire des 4 colonnes selon
    un maximum de 7 lignes d'achats différents.
*/
Random gen = new Random();
int max ;
do{max = gen.Next(8);}while(max==0);
for(int i = 1; i <= 6; i++){
  for(int j = 1; j < max; j++){
          // création d'une nouvelle ligne d'achats
          newRow2 = tabAchats.NewRow();
          newRow2["ClientID"]= 1000+i;
          newRow2["DateAchat"]= new DateTime(2005, gen.Next(12)+1, gen.Next(29)+1);
          newRow2["MontantAchat"] = Math.Round(gen.NextDouble()*1000,2);
          newRow2["Article"]= Enum.GetName(typeof(typArticle),gen.Next(5));
          // Ajouter la ligne à la table Achats
          tabAchats.Rows.Add(newRow2);
  }
  do{max = gen.Next(8);}while(max==0);
}
```

Affichage des données d'un DataSet

Voici affiché dans un composant visuel de NetFramework1.1 et NetFramework 2.0, le DataGrid, la table des Clients avec ses trois colonnes remplies par le programme précédent :

Affichage de la table des Clients

ClientID	ClientNom	Payé
1001	Legrand	☑
1002	Fischer	☑
1003	Dupont	☐
1004	Durand	☐
1005	Lamiel	☑
1006	Renoux	☐

Voici affiché dans un autre composant visuel DataGrid, la table des Achats avec ses quatre colonnes remplies par le programme précédent :

Affichage de la table des Achats (euros HT)

ClientID	Article	MontantAchat	DateAchat
1001	télévison	653,1	24/07/2005
1001	radio	866,27	24/06/2005
1001	cuisinière	58,23	14/02/2005
1002	radio	443,34	12/06/2005
1002	vélo	679,42	10/02/2005
1002	radio	111,33	23/03/2005
1002	vélo	788,25	22/07/2005
1002	cuisinière	877,68	03/07/2005

Le composant DataGrid destiné à afficher une table de données, est remplacé dans la version 2.0 par le DataGridView, mais est toujours présent et utilisable, nous verrons illustrerons l'utilisation d'un DataGridView plus loin dans ce document avec des données provenant d'une Base de Données.

```
private System.Windows.Forms.DataGrid   DataGridSimple;
```

Voici deux façons de lier ce composant visuel à la première table (la table Clients) du DataSet du programme de gestion du Magasin :

```
// Lie directement par le nom de la table le DataGrid au DataSet.
DataGridSimple.SetDataBinding(unDataSet, "Clients");

// Lie indirectement par le rang, le DataGrid au DataSet sur la première table :
DataGridSimple.SetDataBinding(unDataSet, unDataSet.Tables[0].TableName);
```

Enfin pour terminer la description des actions pratiques d'un DataSet, indiquons qu'il est possible de sauvegarder le schéma structuré (relation, clef primaire, tables,…) d'un DataSet dans un fichier de schémas au format **XSL**; il est aussi possible de sauvegarder toutes les données et leur structuration dans un fichier au format **XML**.

Sauvegarde des données du DataSet aux formats XML et XSL :

```
// Stockage uniquement du schéma général du DataSet dans un fichier séparé
unDataSet.WriteXmlSchema("Donnees.xsl");

// Stockage à la fois dans un même fichier du schéma et des données
unDataSet.WriteXml("Donnees.xml", XmlWriteMode.WriteSchema);
```

Fichier "Donnees.xsl" obtenu au format XSL

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:Locale="fr-FR">
```

```
  <xs:complexType>
   <xs:choice maxOccurs="unbounded">
    <xs:element name="Clients">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ClientID" type="xs:int" />
       <xs:element name="ClientNom" type="xs:string" />
       <xs:element name="Payé" type="xs:boolean" minOccurs="0" />
       <xs:element name="Achats" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="ClientID" type="xs:int" minOccurs="0" />
          <xs:element name="Article" type="xs:string" minOccurs="0" />
          <xs:element name="MontantAchat" type="xs:decimal" minOccurs="0" />
          <xs:element name="DateAchat" type="xs:dateTime" minOccurs="0" />
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:choice>
  </xs:complexType>
  <xs:unique name="Constraint1">
   <xs:selector xpath=".//Clients" />
   <xs:field xpath="ClientID" />
  </xs:unique>
  <xs:unique name="Constraint2" msdata:PrimaryKey="true">
   <xs:selector xpath=".//Clients" />
   <xs:field xpath="ClientID" />
   <xs:field xpath="ClientNom" />
  </xs:unique>
  <xs:keyref name="Liste_x0020_des_x0020_achats" refer="Constraint1" msdata:IsNested="true">
   <xs:selector xpath=".//Achats" />
   <xs:field xpath="ClientID" />
  </xs:keyref>
 </xs:element>
</xs:schema>
```

Fichier "Donnees.xml" obtenu format XML

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
 <xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:Locale="fr-FR">
   <xs:complexType>
    <xs:choice maxOccurs="unbounded">
     <xs:element name="Clients">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="ClientID" type="xs:int" />
        <xs:element name="ClientNom" type="xs:string" />
        <xs:element name="Payé" type="xs:boolean" minOccurs="0" />
        <xs:element name="Achats" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="ClientID" type="xs:int" minOccurs="0" />
           <xs:element name="Article" type="xs:string" minOccurs="0" />
```

```
                <xs:element name="MontantAchat" type="xs:decimal" minOccurs="0" />
                <xs:element name="DateAchat" type="xs:dateTime" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>
<xs:unique name="Constraint1">
  <xs:selector xpath=".//Clients" />
  <xs:field xpath="ClientID" />
</xs:unique>
<xs:unique name="Constraint2" msdata:PrimaryKey="true">
  <xs:selector xpath=".//Clients" />
  <xs:field xpath="ClientID" />
  <xs:field xpath="ClientNom" />
</xs:unique>
<xs:keyref name="Liste_x0020_des_x0020_achats" refer="Constraint1" msdata:IsNested="true">
  <xs:selector xpath=".//Achats" />
  <xs:field xpath="ClientID" />
</xs:keyref>
</xs:element>
</xs:schema>
<Clients>
  <ClientID>1001</ClientID>
  <ClientNom>Legrand</ClientNom>
  <Payé>true</Payé>
  <Achats>
    <ClientID>1001</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>456.74</MontantAchat>
    <DateAchat>2005-12-12T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1002</ClientID>
  <ClientNom>Fischer</ClientNom>
  <Payé>true</Payé>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>radio</Article>
    <MontantAchat>297.58</MontantAchat>
    <DateAchat>2005-02-25T00:00:00.0000000+01:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>télévison</Article>
    <MontantAchat>715.1</MontantAchat>
    <DateAchat>2005-07-19T00:00:00.0000000+02:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>télévison</Article>
    <MontantAchat>447.55</MontantAchat>
    <DateAchat>2005-08-16T00:00:00.0000000+02:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1002</ClientID>
```

```
    <Article>cuisinière</Article>
    <MontantAchat>92.64</MontantAchat>
    <DateAchat>2005-09-23T00:00:00.0000000+02:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>171.07</MontantAchat>
    <DateAchat>2005-01-23T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1003</ClientID>
  <ClientNom>Dupont</ClientNom>
  <Payé>false</Payé>
  <Achats>
    <ClientID>1003</ClientID>
    <Article>aspirateur</Article>
    <MontantAchat>445.89</MontantAchat>
    <DateAchat>2005-02-11T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1004</ClientID>
  <ClientNom>Durand</ClientNom>
  <Payé>false</Payé>
  <Achats>
    <ClientID>1004</ClientID>
    <Article>télévison</Article>
    <MontantAchat>661.47</MontantAchat>
    <DateAchat>2005-11-15T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1005</ClientID>
  <ClientNom>Lamiel</ClientNom>
  <Payé>true</Payé>
</Clients>
<Clients>
  <ClientID>1006</ClientID>
  <ClientNom>Renoux</ClientNom>
  <Payé>false</Payé>
  <Achats>
    <ClientID>1006</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>435.17</MontantAchat>
    <DateAchat>2005-06-22T00:00:00.0000000+02:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1006</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>491.3</MontantAchat>
    <DateAchat>2005-12-25T00:00:00.0000000+01:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1006</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>388.81</MontantAchat>
    <DateAchat>2005-10-13T00:00:00.0000000+02:00</DateAchat>
  </Achats>
```

```
  <Achats>
    <ClientID>1006</ClientID>
    <Article>radio</Article>
    <MontantAchat>864.93</MontantAchat>
    <DateAchat>2005-07-23T00:00:00.0000000+02:00</DateAchat>
  </Achats>
 </Clients>
</NewDataSet>
```

ADO .Net et SQL serveur 2005

Plan général:

1. Rappel sur l'installation de SQL Serveur 2005

2. Accès lecture en mode connecté à une BD

Exemple de lecture en mode connecté dans une BD Access

Exemple de lecture en mode connecté dans une BD SQL serveur 2005

mode déconnecté : Affichage avec le DataGridView

1. **DataGridView** lié directement en lecture et écriture à un DataTable
2. Construire les lignes et les colonnes d'un **DataGridView** par programme
3. **DataGridView** lié à un **DataSet**

mode déconnecté : modifications de données à partir d'un DataGridView lié à un DataSet lui-même connecté à une BD

Exercices :

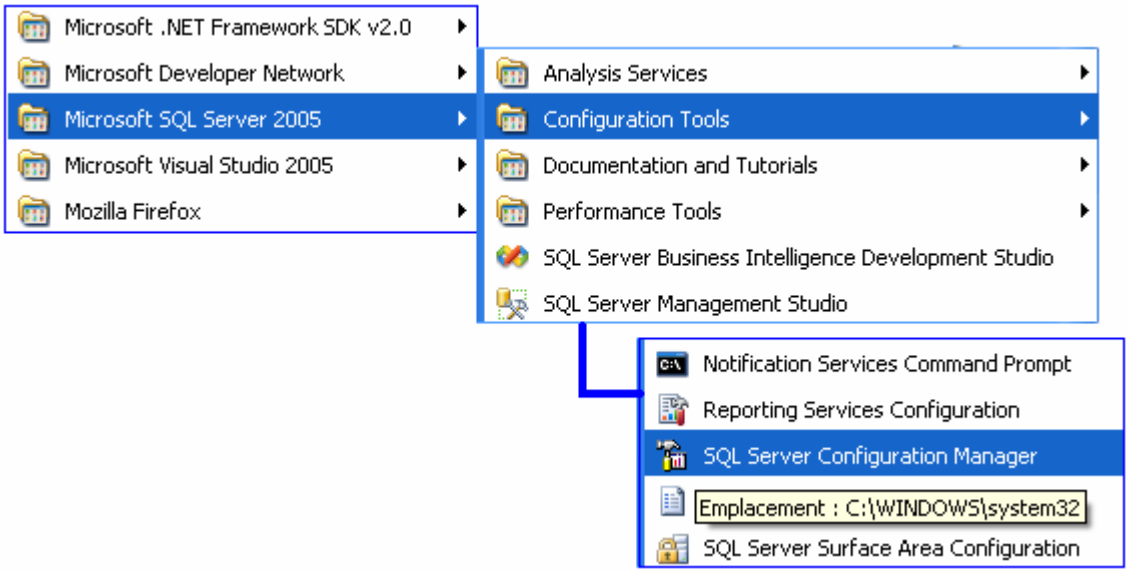
- **Gestion simplifiée d'un petit magasin**
- **Amélioration de la gestion du petit magasin : clef étrangère et delete/update en cascade**

1. Rappel sur l'installation de SQL Serveur 2005

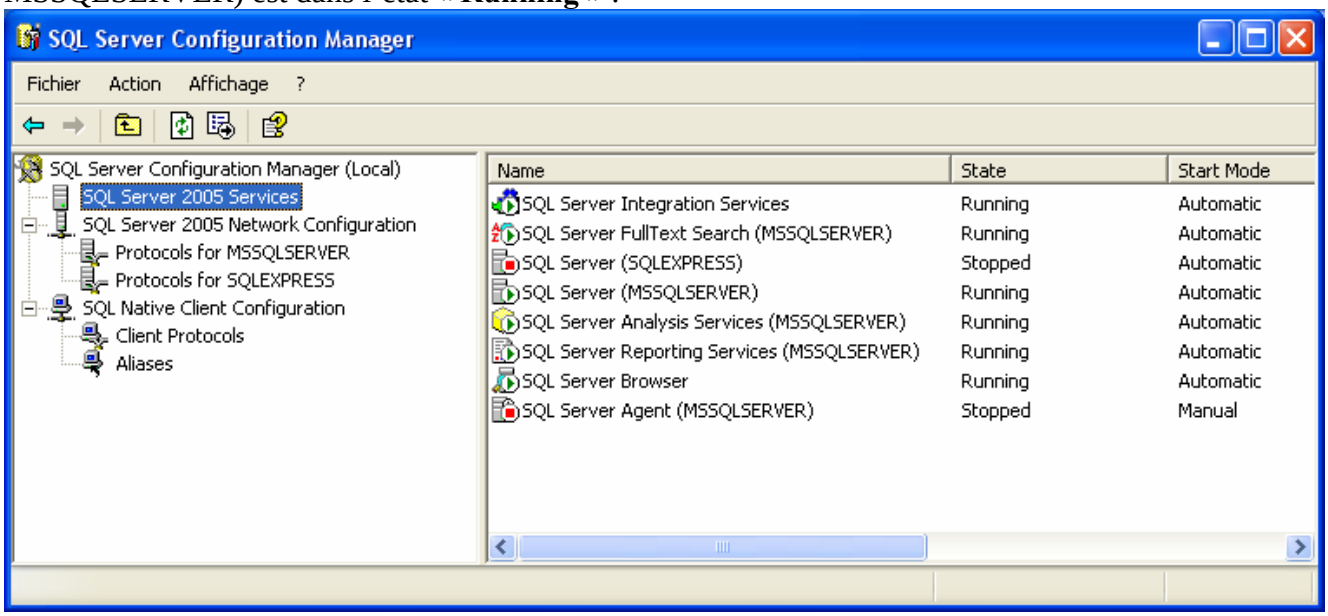
Installation de SQL server 2005 :

- désinstaller toute version précédente de SQL server
- installer SQL server 2005 éd. Developer (2 x CD-ROM).

1°) Vérifiez SQL server 2005 en lançant « **SQL server configuration manager** » :

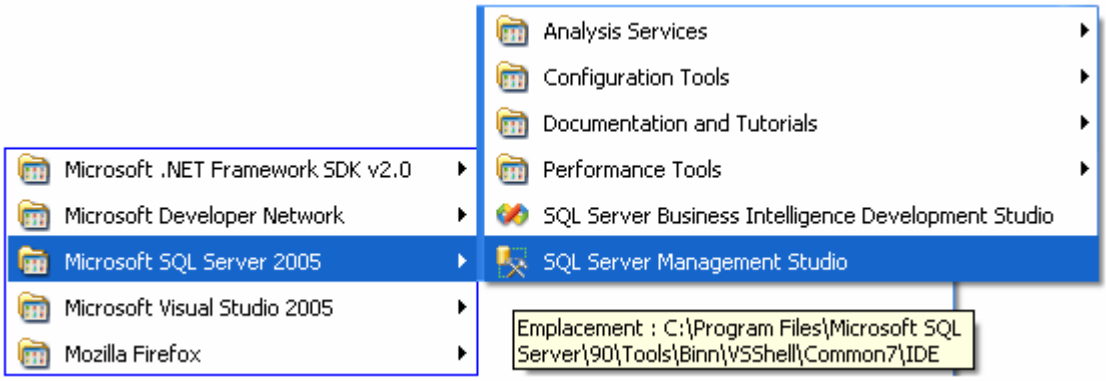


Le gestionnaire de configuration doit vous indiquer que SQL server (nommé ici MSSQLSERVER) est dans l'état « **Running** » :



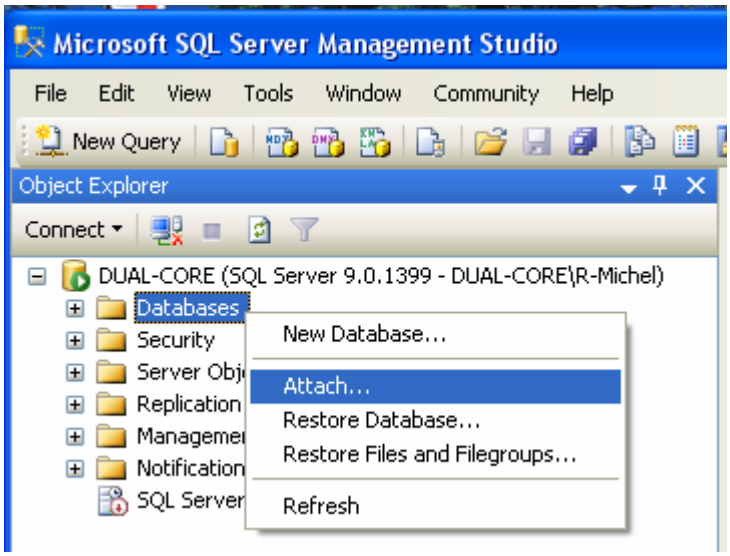
La figure précédente montre qu'il est possible d'installer sur la même machine "SQL server express" (version gratuite) et SQL server 2005 (version payante).

2°) Accéder à une BD SQL server 2005 en lançant « **SQL server Management Studio** » :

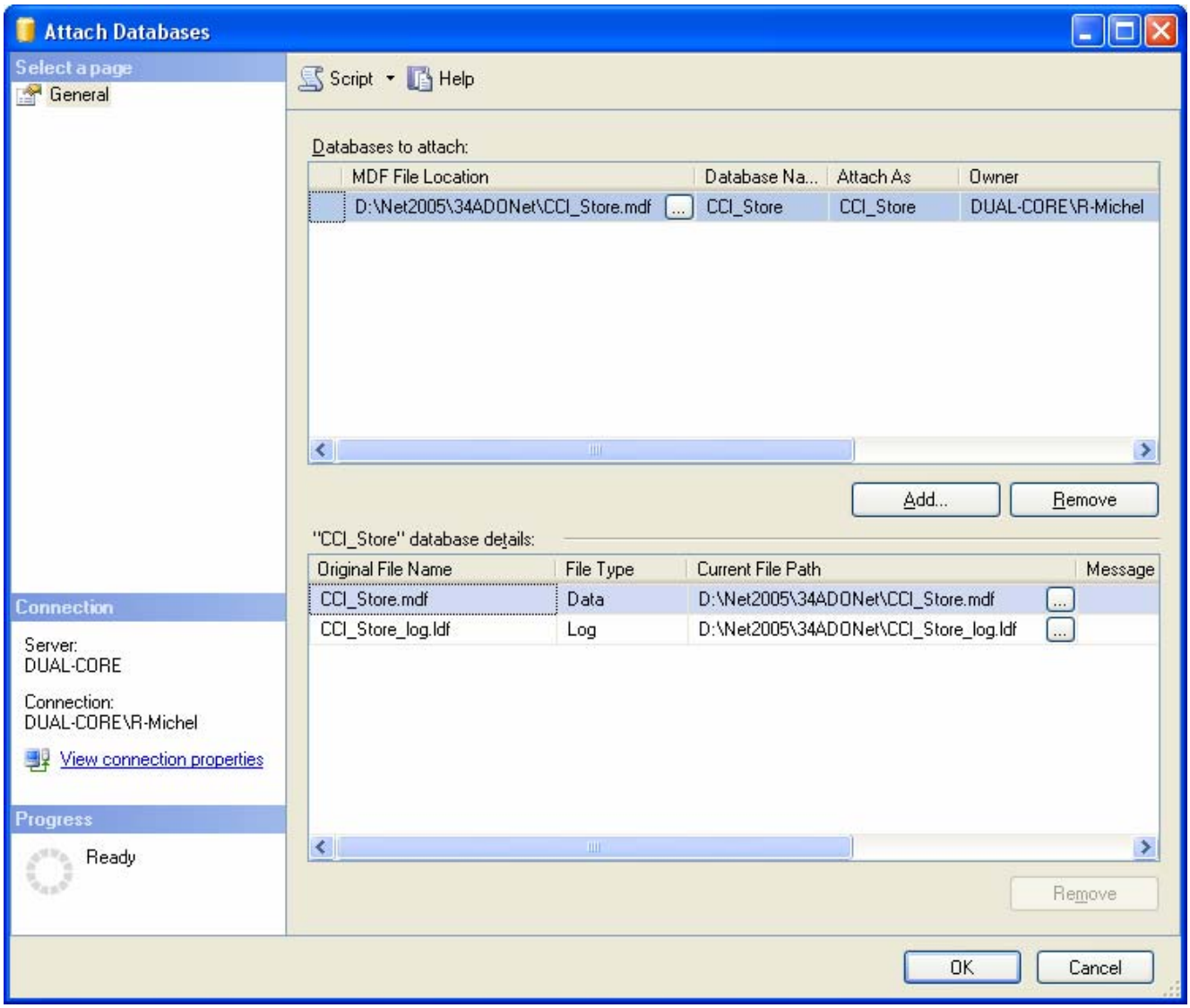


Invite de connexion à suivre ….

3°) Attacher une BD type *.mdf à SQL server 2005 (bouton droit de souris sur Databases) :



Sélectionner le fichier de la BD déjà existante (ici « CCI_Store.mdf ») :

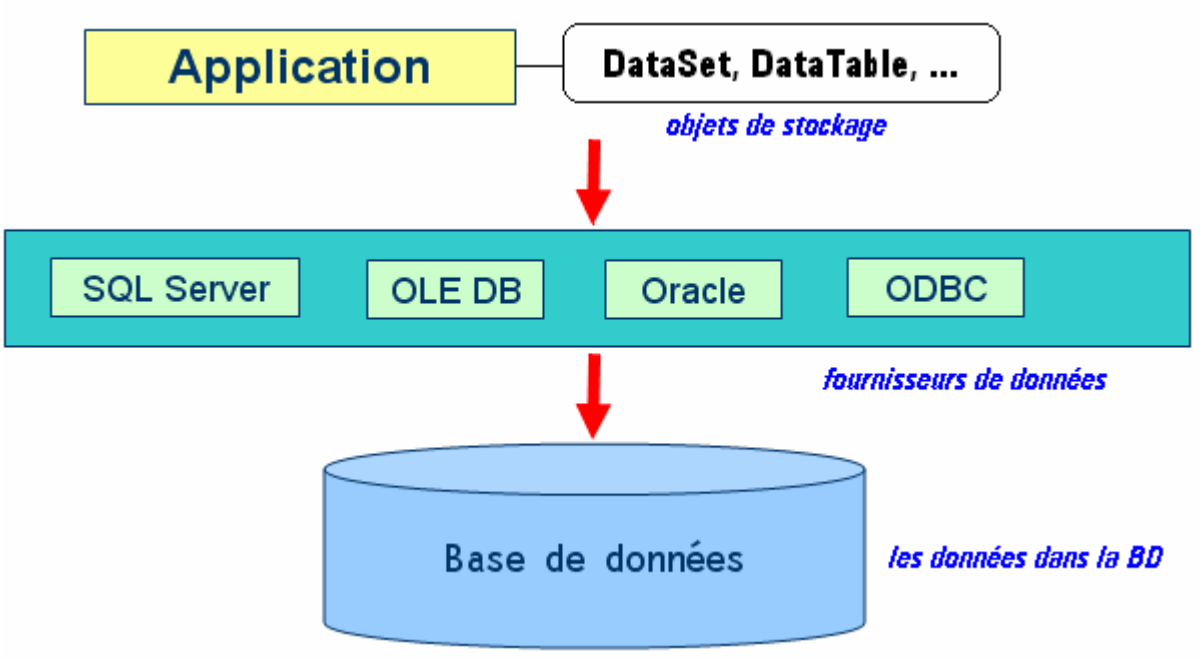


Pour manipuler SQL serveur 2005, voir le manuel fournit par Microsoft ou bien des ouvrages spécialisés sur le sujet. Par la suite, après avoir installé SQL serveur 2005 sur notre machine, nous avons pour objectif de montrer comment accéder à une base de données en local à partir d'un programme écrit en C# en utilisant les outils d'ADO.Net.

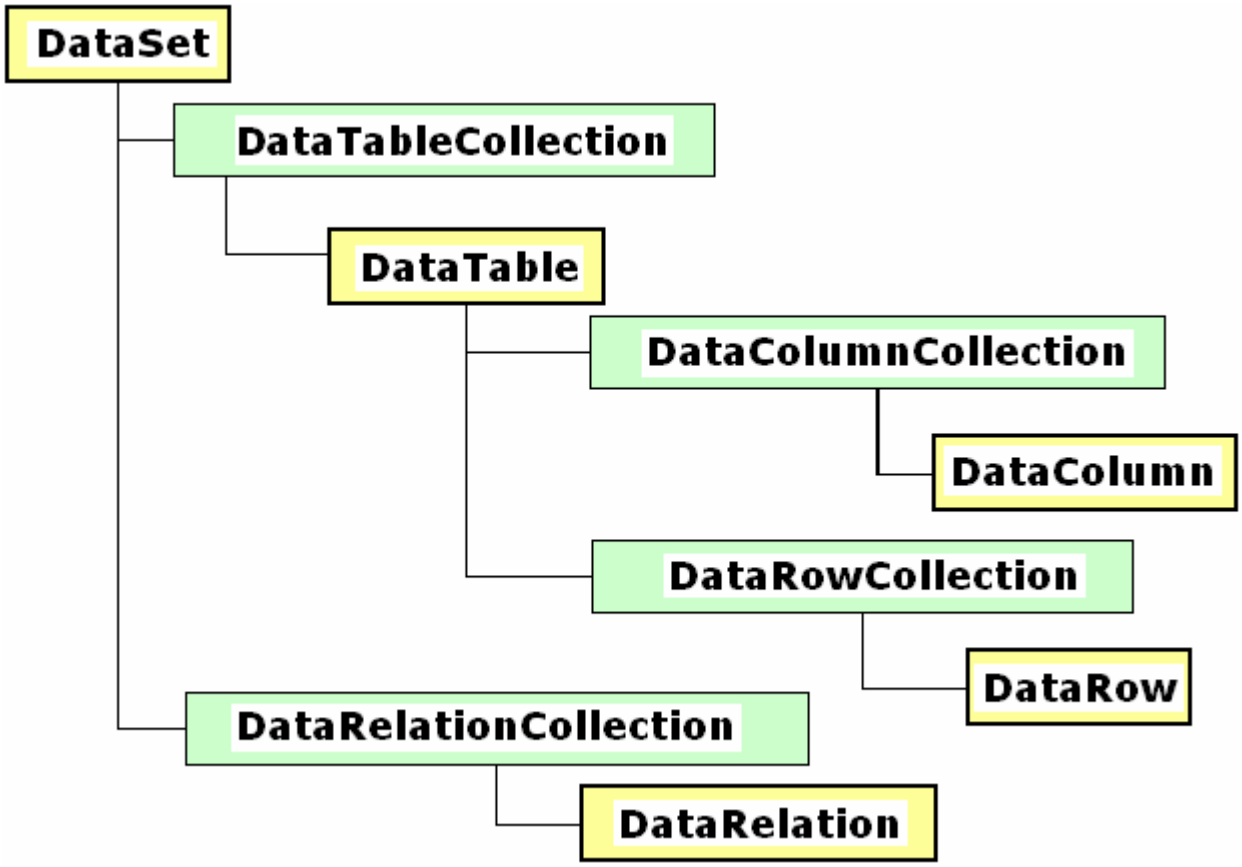
2. Accès lecture en mode connecté à une BD

Rappelons qu'ADO.Net 2.0 est composé de deux ensembles d'outils principaux de gestion des données :

- Les **objets de stockage de données** : objets déconnectés stockant les données sur la machine locale.
- Les **objets fournisseurs** (objets connectés directement) : gérant la communication entre le programme et la base de données.



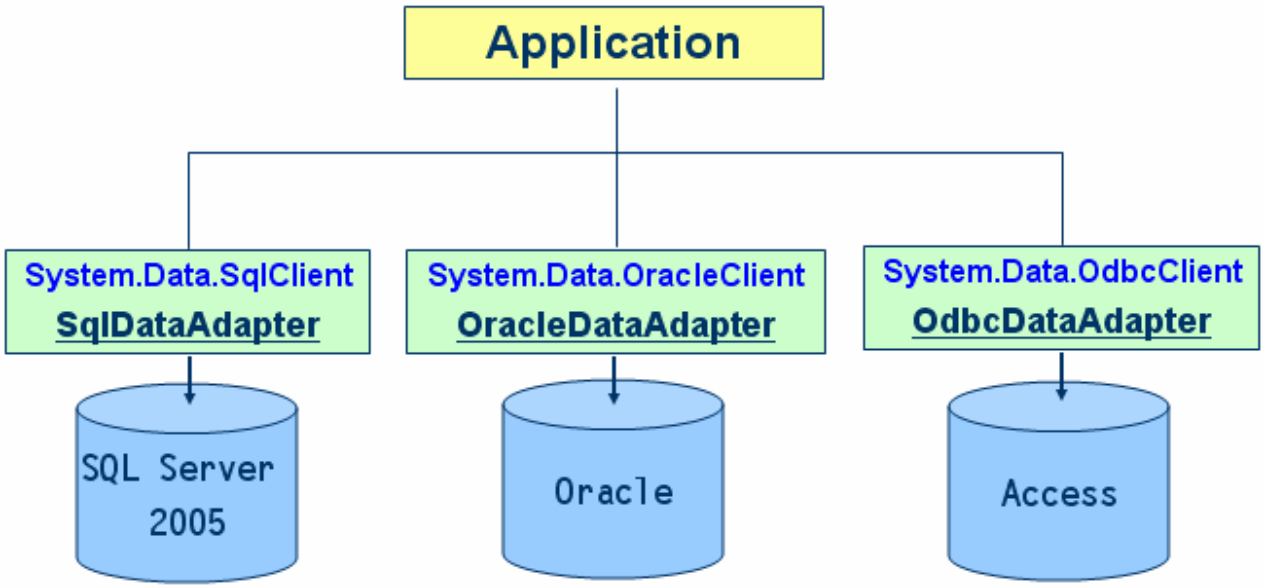
Nous avons exploré dans le chapitre précédent les possibilités offertes par les objets de données comme principalement le DataSet dont nous rappelons l'organisation ci-dessous :



Nous explorons dans ce chapitre, les objets du type fournisseurs et leurs utilisations dans un programme.

Ado .Net met propose plusieurs classes de **fournisseurs** permettant de gérer l'accès à diverses sources de données, toutes situées dans le namespace **System.Data** :

- **System.Data.SQLClient** : accès aux bases de données SQL Serveur
- **System.Data.Odbc** : accès aux bases de données gérées par Odbc
- **System.Data.OracleClient** : accès aux bases de données Oracle
- …..

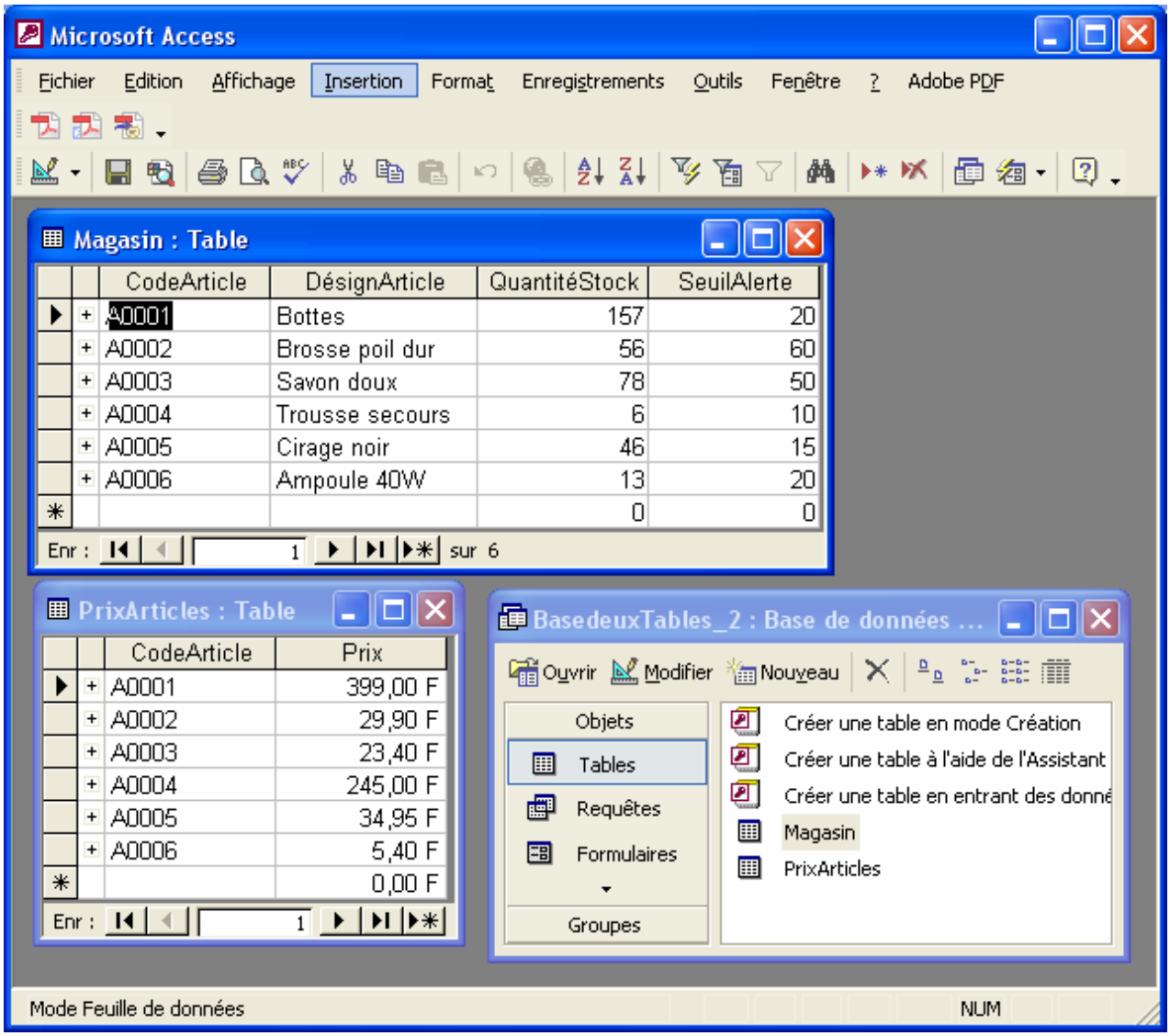


Ci-dessous les principaux objets d'ADO.Net 2.0 en liaison avec les BD :

Objets	Fonctionnalités
OdbcConnection **SqlConnection** **OracleConnection** **……**	Assurent la gestion des connexions physiques vers la base de données du type concerné (ODBC, SQL server, Oracle, ….)
OdbcCommand **SqlCommand** **OracleCommand** **……**	Assure le fonctionnement des commandes SQL (SELECT, INSERT, UPDATE, DELETE) vers la base de données. L'exécution de la commande (aussi appelée requête) s'effectue grâce à 4 méthodes qui se dénomment **Execute**xxx selon le type de traitement à effectuer (par ex : ExecuteReader sert au mode connecté).
OdbcDataReader **SqlDataReader** **OracleDataReader** **……**	Mode connecté : permet de lire les résultats des requêtes renvoyés par l'exécution de la commande SQL (en lecture seule). Ce mode de travail dans lequel le programme est connecté directement sur la BD, est déconseillé dès que l'application est complexe (par exe : modifications fréquentes).
DataSet	Mode déconnecté : permet de lire et de travailler les résultats des requêtes renvoyés par l'exécution de la commande SQL indépendamment de la BD. Le programme effectue un accès à la BD et range les données et leurs relations stockées en mémoire centrale dans un objet DataSet en cache. Ce mode est conseillé car l'application travaille alors sur une "image" des données et de leurs relations. Il faut ensuite effectuer une validation des modifications apportées aux données dans le cache.

Exemple de lecture en mode connecté dans une BD Access

Lancement de la lecture d'une table " Magasin " dans une BD Access nommée "BasedeuxTables_2.mdb":



Nous nous prioposons d'écrire un programme lisant et affichant le contenu de cette BD, enregistrement par enregistrement.

Le schéma général qui est adopté pour une telle action est le suivant :

- **Création d'un objet de connexion physique à la BD.**
- **Ouverture de la connexion physique à la BD.**
- **Création d'un objet de commande avec une requête.**
- **Exécution de la commande en mode connecté.**
- **Lecture et affichage des champs résultats de la commande.**
- **Fermeture de la connexion physique à la BD.**
- **Libération des ressources utilisées.**

Code source d'un programme affichant les résultats d'une requête dans un textBox :

```
// chemin vers la BD (Access):
string urlAcess = "Driver={Microsoft Access Driver (*.mdb)};"
  + @"DBQ=C:\BasedeuxTables_2.mdb";

// objet de connexion :
OdbcConnection connexion;
connexion = new OdbcConnection(urlAcess);

try
{
    // ouvrir la connexion :
    connexion.Open();

    // préparation de la requête effective :
    string Requete = "SELECT * FROM Magasin";
    OdbcCommand cmdAcess = new OdbcCommand(Requete, connexion);

    // lecture de plusieurs lignes :
    OdbcDataReader lignesRead;
    lignesRead = cmdAcess.ExecuteReader(CommandBehavior.CloseConnection);
    while (lignesRead.Read())
    {
      textBoxAffiche.AppendText(lignesRead.GetString(0) + " " +
        lignesRead.GetString(1) + " " +
        lignesRead.GetString(2) + " " +
        lignesRead.GetString(3) + "\r\n");
    }

  // fermer la connexion :
   connexion.Close();
    }
catch (Exception ex)
{
  Console.WriteLine("Désolé : " + ex.Message);
   textBoxAffiche.Clear();
   textBoxAffiche.AppendText("Désolé : " + ex.Message);
 }

    // libérer les ressources :
    connexion.Dispose();
    connexion = null;
```

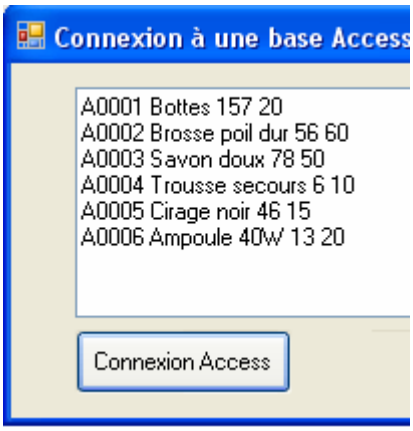
Création d'un objet de connexion physique à la BD : BasedeuxTables_2.mdb

Ouverture de la connexion physique à la BD.

Création d'un objet de commande avec une requête.

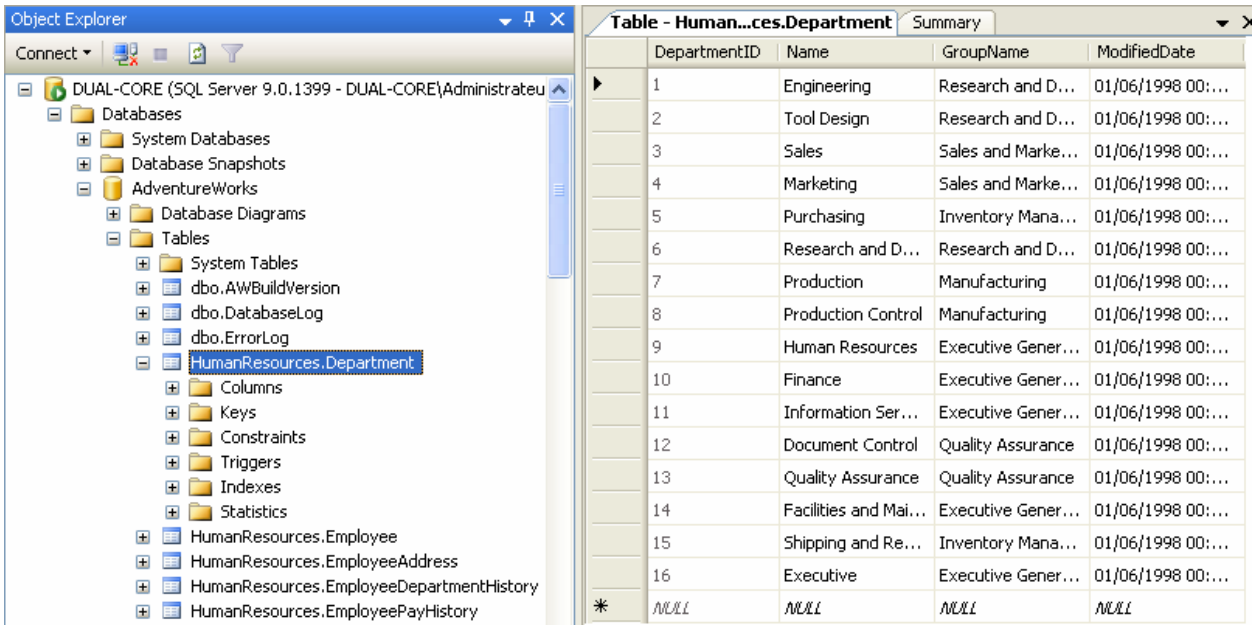
Exécution de la commande en mode connecté.

Lecture et affichage des 4 champs résultats de la commande.



Exemple de lecture en mode connecté dans une BD SQL serveur 2005

Reprenons la même action sur la BD de démonstration gratuite fournie par Microsoft avec SQL serveur 2005 "AdventureWorks.mdf " et lançons la lecture des 16 enregistrements de la table "HumanResources.Departement":




```
// chemin vers la BD (SQL serveur 2005):
string urlSqlServer = @"Data Source=(local);Initial Catalog=AdventureWorks;" + "Integrated Security=SSPI;";
```

Le mécanisme reste le même que celui qui a été utilisé pour la BD Access, seuls les objets permettant la connexion, la commande, la lecture sont différents mais opèrent d'une manière identique. Ci-dessous les lignes de code qui diffèrent selon le type de BD utilisée :

```
// objet de connexion :
SqlConnection connexion;
connexion = new SqlConnection(urlSqlServer);
```

….

```
// préparation de la requête effective :
string Requete = "SELECT * FROM HumanResources.Department";
SqlCommand cmdSqlServ = new SqlCommand (Requete, connexion);
```

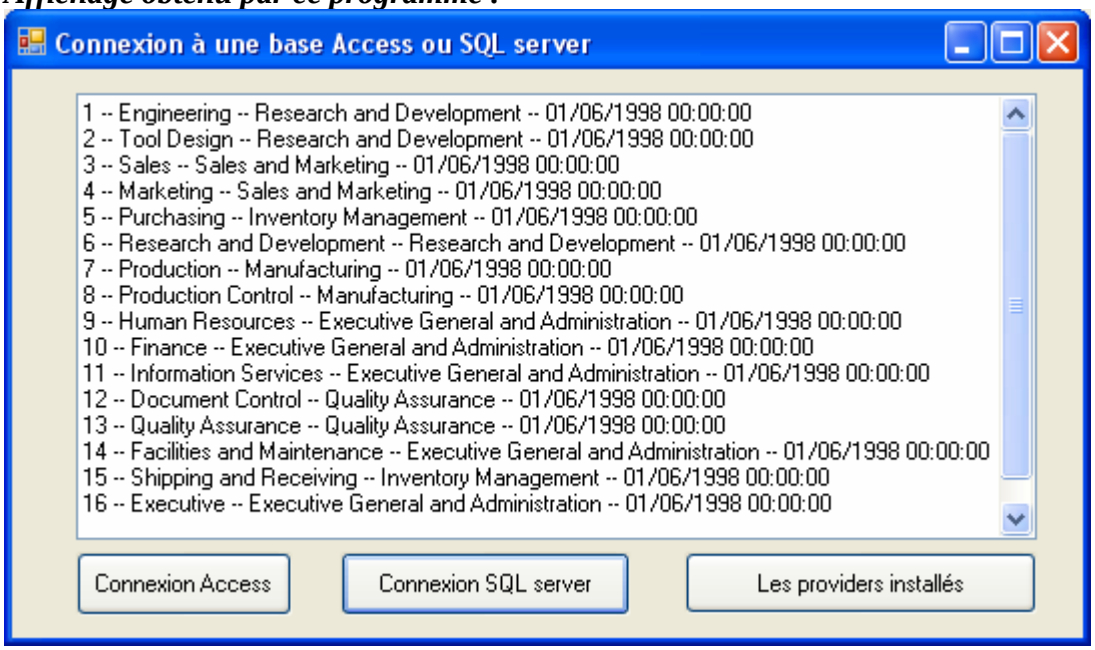
….

```
//lecture de plusieurs lignes :
SqlDataReader lignesRead;
lignesRead = cmdSqlServ.ExecuteReader(CommandBehavior.CloseConnection);
textBoxAffiche.Clear();
while (lignesRead.Read())
{
    textBoxAffiche.AppendText(Convert.ToString(lignesRead.GetInt16(0)) + " -- " +
        lignesRead.GetString(1) + " -- " +
        lignesRead.GetString(2) + " -- " +
        lignesRead.GetDateTime(3).ToString() + "\r\n");
}
```

Code source complet du programme affichant les résultats de la requête dans un textBox :

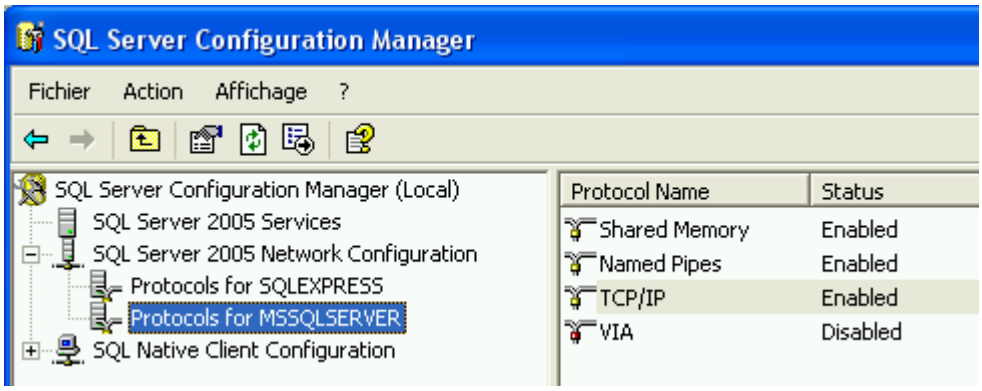
```
// chemin vers la BD (SQL serveur 2005):
string urlSqlServer = @"Data Source=(local);Initial Catalog=AdventureWorks;" + "Integrated Security=SSPI;";

// objet de connexion :
SqlConnection connexion;
connexion = new SqlConnection(urlSqlServer);

Console.WriteLine();

// ouvrir la connexion :
connexion.Open();

// traitement de la BD :
string Requete = "SELECT * FROM HumanResources.Department";
SqlCommand cmdSqlServ = new SqlCommand(Requete, connexion);

//lecture de plusieurs lignes :
SqlDataReader lignesRead;
lignesRead = cmdSqlServ.ExecuteReader(CommandBehavior.CloseConnection);
textBoxAffiche.Clear();
while (lignesRead.Read())
{
	textBoxAffiche.AppendText(Convert.ToString(lignesRead.GetInt16(0)) + " -- " +
		lignesRead.GetString(1) + " -- " +
		lignesRead.GetString(2) + " -- " +
		lignesRead.GetDateTime(3).ToString() + "\r\n");
}

// fermer la connexion :
connexion.Close();

// libérer les ressources :
connexion.Dispose();
connexion = null;
```

Affichage obtenu par ce programme :

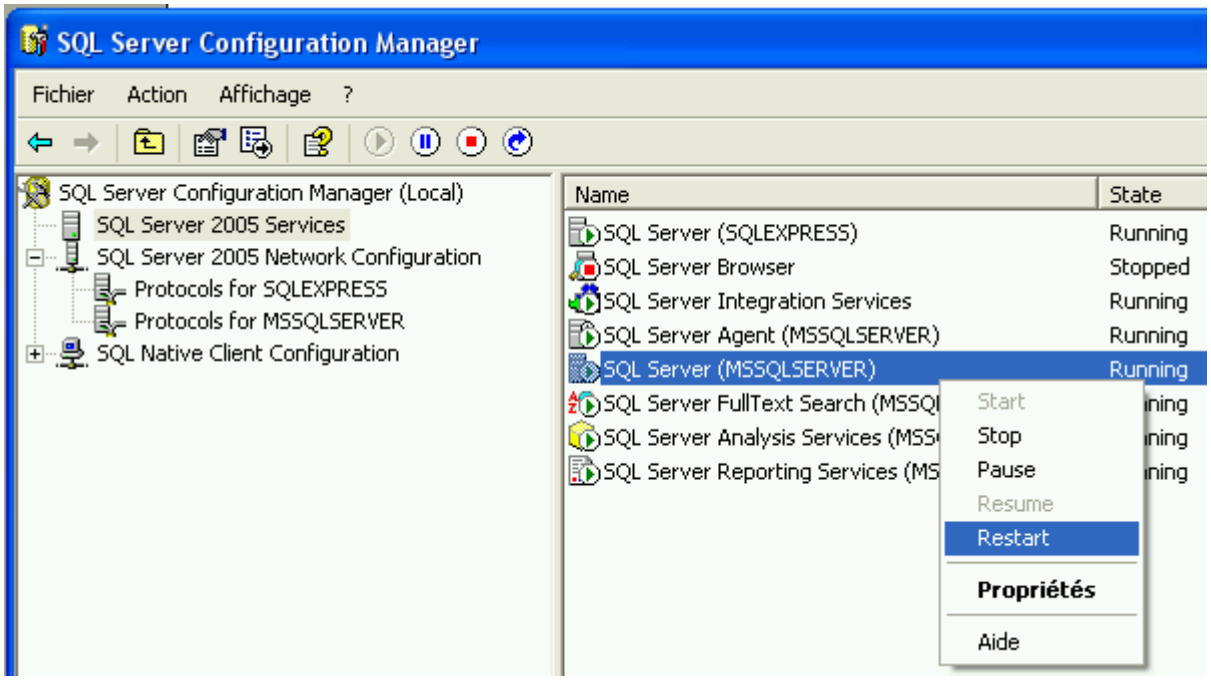


Remarque pratique de paramétrage de SQL serveur 2005 :

Si dans une utilisation pratique des exemples traités ci-après vous obtenez un message d'erreur indiquant que la connexion TCP/IP n'a pas pu avoir lieu, vérifiez que le statut du protocole TCP/IP est « **enabled** » en exécutant le programme SQL Server Config.Manager de SQL serveur 2005 :



N'oubliez pas ensuite de redémarrer SQL Server, à partir du même SQL Server Config. :



mode déconnecté : Affichage avec le DataGridView

Un **DataGridView** permet d'afficher des données provenant de différentes sources grâce à sa propriété de liaison de données (data-bind) DataSource en lecture ou écriture, qui est une référence vers l'objet qui contient les données à afficher.

1°) On peut lier un **DataGridView** directement en lecture et écriture à un DataTable :

```
DataGridView dataGridView1;.........

// création d'un DataTable :
DataTable table = new DataTable("Personnes");
table.Columns.Add("Numéro", typeof(string));
table.Columns[0].Unique = true;
table.Columns.Add("Nom", typeof(string));
table.Columns.Add("Prénom", typeof(string));
table.Columns.Add("age", typeof(byte));
table.Columns.Add("Revenus", typeof(double));

/* Data-bind du dataGridView1:
* après cette instruction le dataGridView1 est lié à la table :
* toute ligne entrée manuellement dans le dataGridView1 est
* automatiquement ajoutée dans la table.
* */
dataGridView1.DataSource = table;//on relie à l'objet table
```

	Numéro	Nom	Prénom	age	Revenus
▸*					

//affichage de la table (vide pous l'instant)

```
/* Réciproquement, toute donnée ajoutée à la table est immédiatement
* visualisée dans le dataGridView1.
* */
table.Rows.Add(888, "fffff", "xxxxx", 88, 8888);
table.Rows.Add(999, "ggg", "yyyy", 99, 9999);
```

	Numéro	Nom	Prénom	age	Revenus
	888	fffff	xxxxx	88	8888
	999	ggg	yyyy	99	9999
▸*					

//affichage de la table (avec les deux lignes qui viennent d'être ajoutées)

2°) On peut construire les lignes et les colonnes d'un **DataGridView** par programme :

```
DataGridView dataGridView1;.........

// chargement par programme du dataGridView :
dataGridView1.ColumnCount = 5;
dataGridView1.Columns[0].Name = "un";
```

```
dataGridView1.Columns[1].Name = "deux";
dataGridView1.Columns[2].Name = "trois";
dataGridView1.Columns[3].Name = "quatre";
dataGridView1.Columns[4].Name = "cinq";
dataGridView1.Rows.Add(123, "aaa", "bbb", 25, 1235.58);
dataGridView1.Rows.Add("xxxx", true, -23.6, "rrrr", 45);
dataGridView1.Rows.Add(4);
/* Remarque : le type n'est pas fixé tout est converti en string. */
```

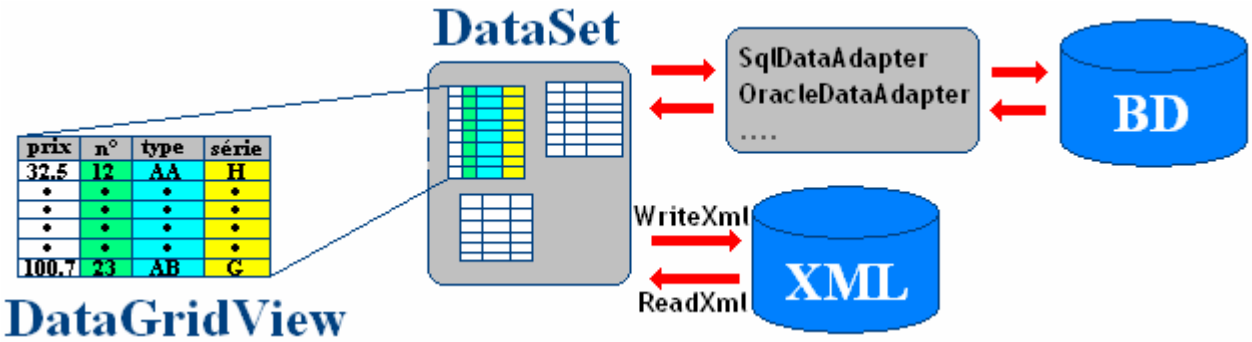
	un	deux	trois	quatre	cinq
▶	123	aaa	bbb	25	1235,58
	xxxx	True	-23,6	rrrr	45

//affichage des lignes et des colonnes entrées par programme

3°) **DataGridView** lié à un **DataSet** :

Le principe d'utilisation des outils déconnectés est simple avec ADO.net, il est fondé sur une architecture MVC (modèle-vue-contrôleur) dans laquelle un **DataSet** est connecté à une BD et gère le modèle des données en mémoire centrale, un **DataGridView** est alors relié à une table du **DataSet**, il est ainsi chargé de gérer la vue et les interactions événementielles avec l'utilisateur.

Un **DataSet**, comme nous l'avons déjà vu dans un chapitre précédent permet de lire et d'écrire directement des données au format XML, il permet aussi par l'intermédiaire d'un objet du type **SqlDataAdapter** ou **OdbcDataAdapter** ou **OracleDataAdapter**…, de lire et d'écrire dans une BD de type SQL serveur, ODBC, Oracle… Le **DataGridView** sert alors à afficher et à modifier les données d'une table du **DataSet** :



D'une manière générale, un **xxxDataAdapter** est utilisé lors d'un échange avec un **DataSet** ou un **DataTable** (d'un **DataSet** ou non), un objet **xxxDataAdapter** est chargé d'assurer la liaison entre la BD physique et le **DataSet** ou le **DataTable** :

- Il contient des propriétés de commande SQL :
  - **public** xxxCommand SelectCommand{get; set;},
  - **public** xxxCommand InsertCommand {get; set;},
  - **public** xxxCommand UpdateCommand {get; set;},
  - **public** xxxCommand DeleteCommand {get; set;}, permettant de lancer ces commandes à partir du **DataTable** sur la BD.

- Il contient réciproquement des méthodes permettant de charger un **DaTable** ou un **DataSet** contenant un ou plusieurs **DataTable** (**public** int Fill(DataTable datTable), **public** int Fill(DataSet datSet),….)

Un **xxxDataAdapter** ouvre et ferme la connexion automatiquement (Open et Close).

DataGridView lié à un DataSet avec fichier XML

Nous montrons ici comment sauvegarder tout le contenu d'un DataGridView nommé dataGridView1 dans un fichier nommé "Table.xml" en utilisant un DataSet nommé dsTable.

a) Utilisons le DataTable nommé "Personnes" que nous avons créé précédemment :

```
DataTable table = new DataTable("Personnes");
table.Columns.Add("Numéro", typeof(string));
table.Columns[0].Unique = true;
table.Columns.Add("Nom", typeof(string));
table.Columns.Add("Prénom", typeof(string));
table.Columns.Add("age", typeof(byte));
table.Columns.Add("Revenus", typeof(double));
table.Rows.Add(888, "fffff", "xxxxx", 88, 8888);
table.Rows.Add(999, "ggg", "yyyy", 99, 9999);
```

b) Ajoutons cette table "Personnes" au DataSet dsTable :

```
DataSet dsTable = new DataSet();// création d'un DataSet
dsTable.Tables.Add(table); //ajout de la table "Personnes" au DataSet
```

c) Le dataGridView1 est lié au DataSet nommé dsTable par sa propriété DataSource et plus précisément à la première table du DataSet par sa propriété DataMember :

```
dataGridView1.DataSource = dsTable;
dataGridView1.DataMember = dsTable.Tables[0].TableName;
```

d) Le DataSet posède des méthodes permettant de sauvegarder les données brutes au format XML (méthode **WriteXml**), il peut aussi sauvegarder sa structure générale sous forme d'un schéma au format XML (méthode **WriteXmlSchema**). On peut dès lors, sauvegarder tout le contenu du dataGridView1 dans un fichier nommé "Table.xml" et le schéma du DataSet dans un fichier nommé " schemaTable.xml", en utilisant les méthodes **WriteXml** et **WriteXmlSchema** :

```
dsTable.WriteXmlSchema("schemaTable.xml");
dsTable.WriteXml("Table.xml");
```

Code d'un click sur un bouton permettant de sauvegarder au format XML :

```
DataSet dsTable = new DataSet();// création d'un DataSet
```

Button buttonSaveDataSet :

// bouton de sauvegarde au format XML

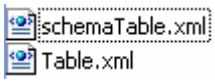
```
private void buttonSaveDataSet_Click(object sender, EventArgs e)
{
 // sauvegarde au format XML par DataSet du contenu du dataGridView :
 if (dsTable.Tables.Count != 0)// si le DataSet contient quelque chose !
 {
   dataGridView1.DataSource = dsTable;
   dataGridView1.DataMember = dsTable.Tables[0].TableName;
   dsTable.WriteXmlSchema("schemaTable.xml");
   dsTable.WriteXml("Table.xml");
   textBox1.Text = "nbr lignes = " +
     dsTable.Tables[0].Rows.Count.ToString();
  }
}
```

Le dataGridView1 a été chargé par le code a) , b) et c) :

	Numéro	Nom	Prénom	age	Revenus
	888	fffff	xxxxx	88	8888
	999	ggg	yyyy	99	9999
▸*					

Le click sur le bouton

engendre deux fichiers au format XML :



Voici le contenu du fichier "schemaTable.xml":

```
  <?xml version="1.0" standalone="yes" ?>
- <xs:schema id="Temporaire" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" xm
  - <xs:element name="Temporaire" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
    - <xs:complexType>
      - <xs:choice minOccurs="0" maxOccurs="unbounded">
        - <xs:element name="Personnes">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="Numéro" type="xs:string" minOccurs="0" />
                <xs:element name="Nom" type="xs:string" minOccurs="0" />
                <xs:element name="Prénom" type="xs:string" minOccurs="0" />
                <xs:element name="age" type="xs:unsignedByte" minOccurs="0" />
                <xs:element name="Revenus" type="xs:double" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    - <xs:unique name="Constraint1">
        <xs:selector xpath=".//Personnes" />
        <xs:field xpath="Numéro" />
      </xs:unique>
    </xs:element>
  </xs:schema>
```

On remarquera que la contrainte d'unicité pour les données de la colonne 0 est présente dans le schéma.

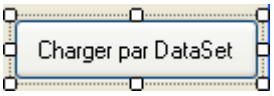
Voici le contenu du fichier"Table.xml" :

```
  <?xml version="1.0" standalone="yes" ?>
- <Temporaire>
  - <Personnes>
      <Numéro>888</Numéro>
      <Nom>fffff</Nom>
      <Prénom>xxxxx</Prénom>
      <age>88</age>
      <Revenus>8888</Revenus>
    </Personnes>
  - <Personnes>
      <Numéro>999</Numéro>
      <Nom>ggg</Nom>
      <Prénom>yyyy</Prénom>
      <age>99</age>
      <Revenus>9999</Revenus>
    </Personnes>
  </Temporaire>
```

Nous montrons maintenant comment récupérer les données et leur structure dans un DataGridView nommé dataGridView1 à partir des fichiers précédents "Table.xml" et "schemaTable.xml".

Le DataSet posède des méthodes permettant de lire les données brutes au format XML (méthode **ReadXml**), et de lire sa structure générale stockée au format XML (méthode **ReadXmlSchema**). On peut dès lors, afficher dans le dataGridView1 le contenu des fichiers nommés "Table.xml" et " schemaTable.xml", en utilisant les méthodes **ReadXml** et **ReadXmlSchema** puis en liant le dataGridView1 à la table du DataSet :

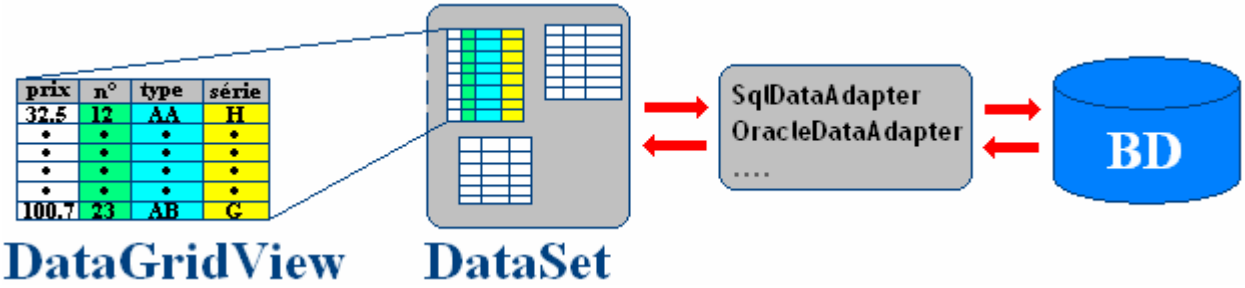
Code d'un click sur un bouton permettant de charger les fichiers XML :

Button buttonLoadDataSet : 

// bouton de chargement du fichier XML

```
private void buttonLoadDataSet_Click(object sender, EventArgs e)
{
 // récupération au format XML par DataSet du contenu du dataGridView :
 RAZ_DataGridView();// on efface
 dsTable = new DataSet("Temporaire");//RAZ du DataSet
 dsTable.ReadXmlSchema("schemaTable.xml");
 dsTable.ReadXml("Table.xml");
 dataGridView1.DataSource = dsTable;
 dataGridView1.DataMember = dsTable.Tables[0].TableName;
}
```

***DataGridView** lié à un **DataSet** connecté à une BD*

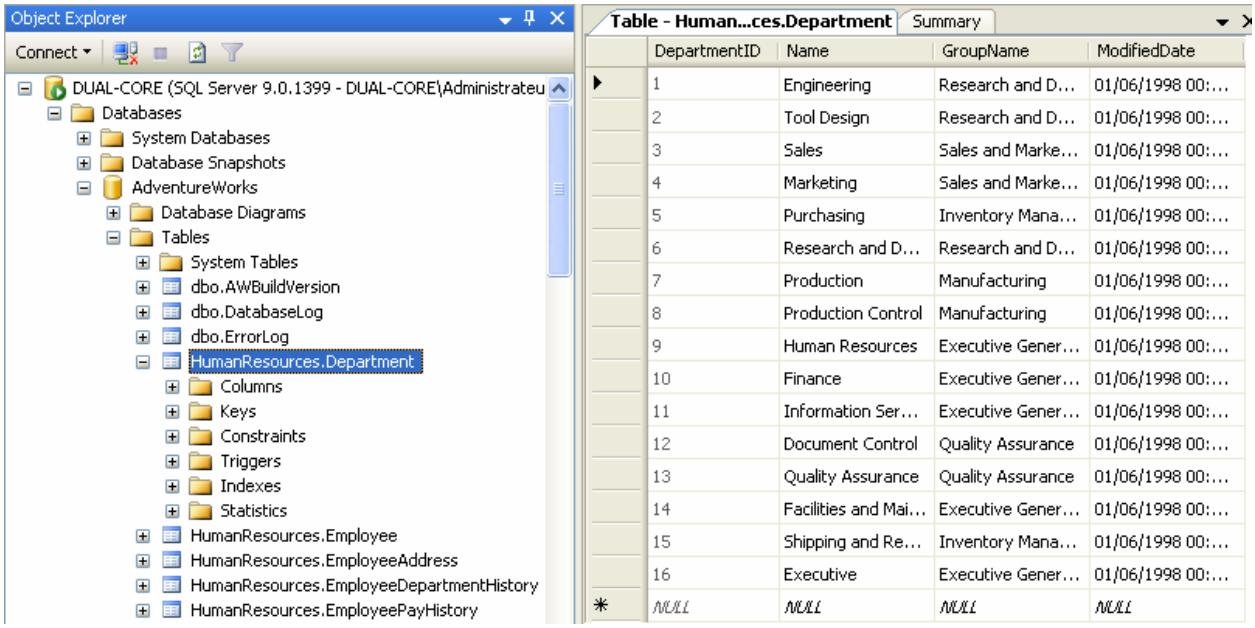


Le principe reste le même que celui qui a été décrit précédemment du moins en ce qui concerne la liaison du DataGridView avec le DataSet.

La connexion aux données ne s'effectue pas directement comme avec des données au format XML, mais à travers un fournisseur de connexion à la BD adapté au type de la BD (SqlDataAdapter pour une BD SQL serveur, OracleDataAdapter pour une BD Oracle, …)

Exemple de code pour affichage de la table "HumanResources.Departement" dans la BD "AdventureWorks.mdf" de Microsoft :

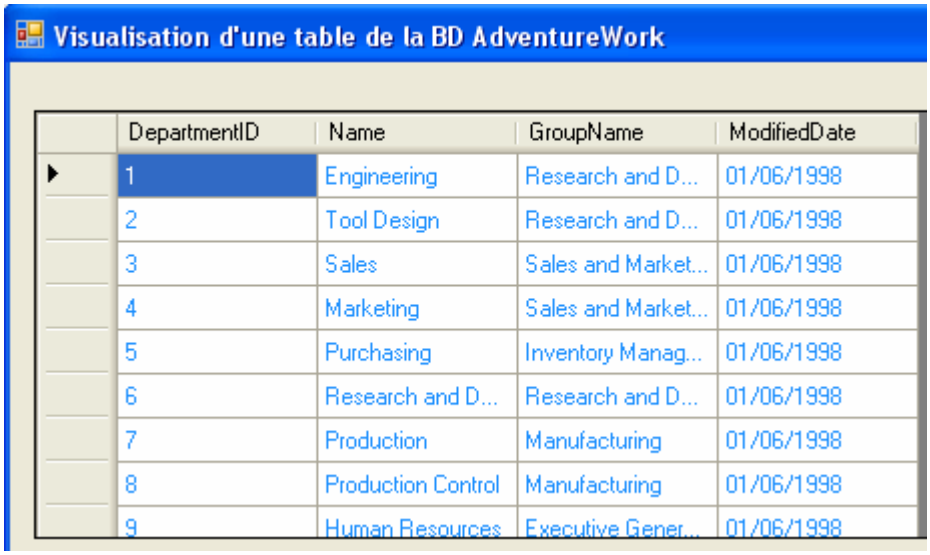
Nous avons précédemment lancé la lecture des 16 enregistrements de cett table en mode connecté, ici nous montrons comment afficher cette même table en mode déconnecté.




```
private void loadDataGridViewFromDataSet()
{/*----- Liaison par DataSet     -----*/

  //--pour SQL server Ed.Developper 2005 :
  string urlSqlServer = @"Data Source=(local);Initial Catalog=AdventureWorks;"
                        +"Integrated Security=SSPI;";

  // objet de connexion :
  SqlConnection connexion = new SqlConnection(urlSqlServer);
```

```
 try
 {
    // unSqDataAdapter : objet de communication et d'échange de données entre DataSet et la BD
    SqlDataAdapter unSqlAdapter = new SqlDataAdapter("Select * From HumanResources.Department ",
connexion);

    //Remplissage du DataSet avec la table Person.Address :
    unSqlAdapter.Fill(unDataSet);

    // Liaison du dataGridView1 à la table du DataSet  :
    dataGridView1.DataSource = unDataSet.Tables[0];

 }
 catch (SqlException ex)
 {
  MessageBox.Show(ex.Message,"Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
 }
}
```

Résultat obtenu dans le DataGridView :

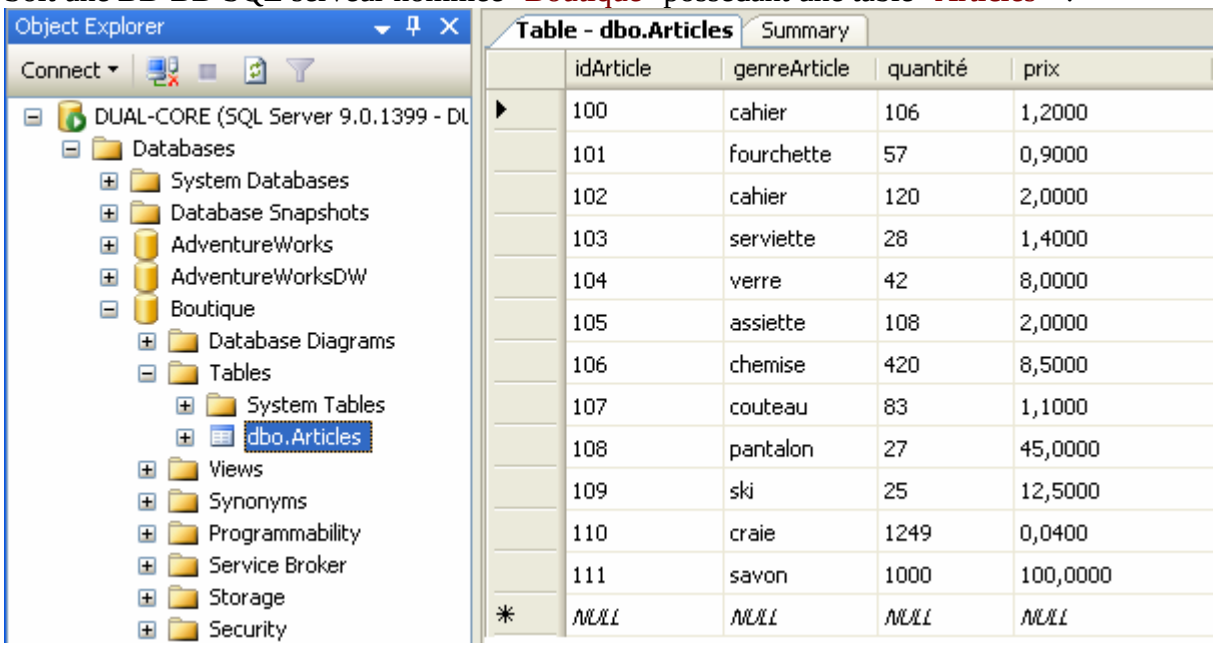


DepartmentID	Name	GroupName	ModifiedDate
1	Engineering	Research and D...	01/06/1998
2	Tool Design	Research and D...	01/06/1998
3	Sales	Sales and Market...	01/06/1998
4	Marketing	Sales and Market...	01/06/1998
5	Purchasing	Inventory Manag...	01/06/1998
6	Research and D...	Research and D...	01/06/1998
7	Production	Manufacturing	01/06/1998
8	Production Control	Manufacturing	01/06/1998
9	Human Resources	Executive Gener...	01/06/1998

mode déconnecté : modifications de données à partir d'un DataGridView lié à un DataSet lui-même connecté à une BD

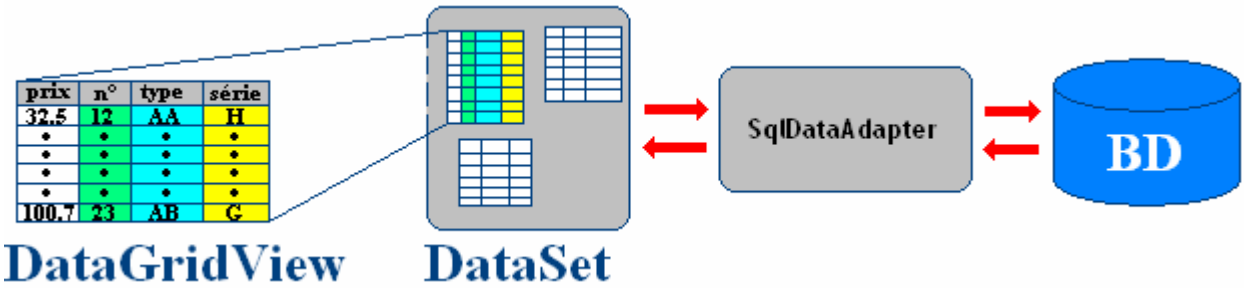
Le principe d'utilisation des outils déconnectés est simple avec ADO.net, il est fondé sur une architecture MVC (modèle-vue-contrôleur), nous proposons un exemple de travail en lecture et écriture dans une BD à partir de modifications effectuées dansun DataGridView lié à cette BD :

Soit une BD BD SQL serveur nommée "Boutique" possédant une table "Articles" :



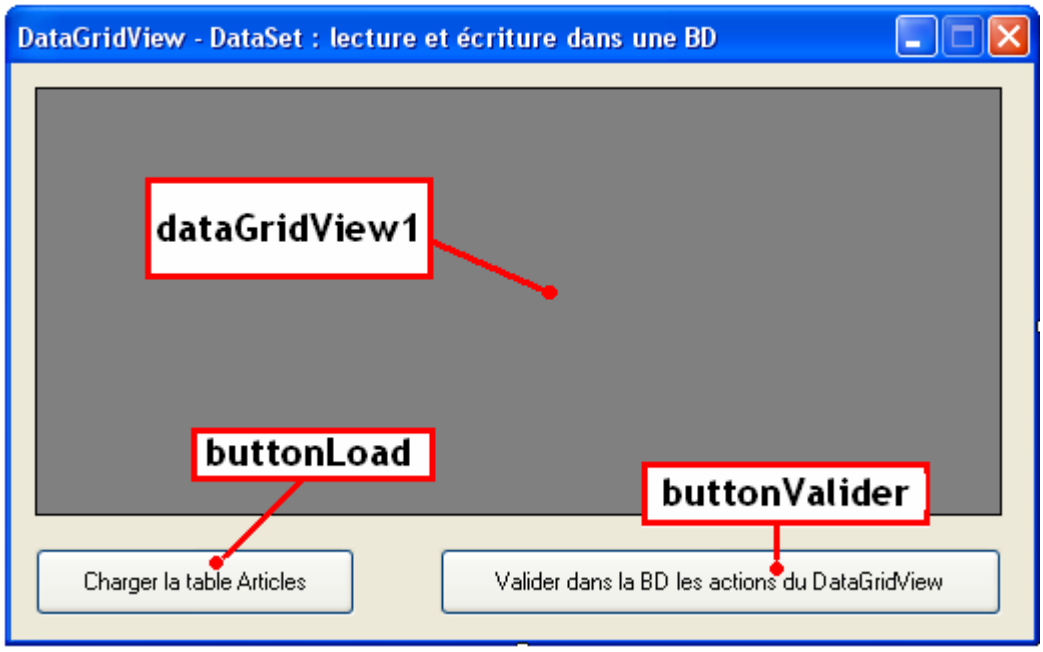
idArticle	genreArticle	quantité	prix
100	cahier	106	1,2000
101	fourchette	57	0,9000
102	cahier	120	2,0000
103	serviette	28	1,4000
104	verre	42	8,0000
105	assiette	108	2,0000
106	chemise	420	8,5000
107	couteau	83	1,1000
108	pantalon	27	45,0000
109	ski	25	12,5000
110	craie	1249	0,0400
111	savon	1000	100,0000
NULL	NULL	NULL	NULL

Un DataSet charge la table "Articles" de la BD SQL serveur nommée "Boutique.mdf" à travers un objet de communication SqlDataAdapter :



On se propose d'écrire une IHM simple contenant :

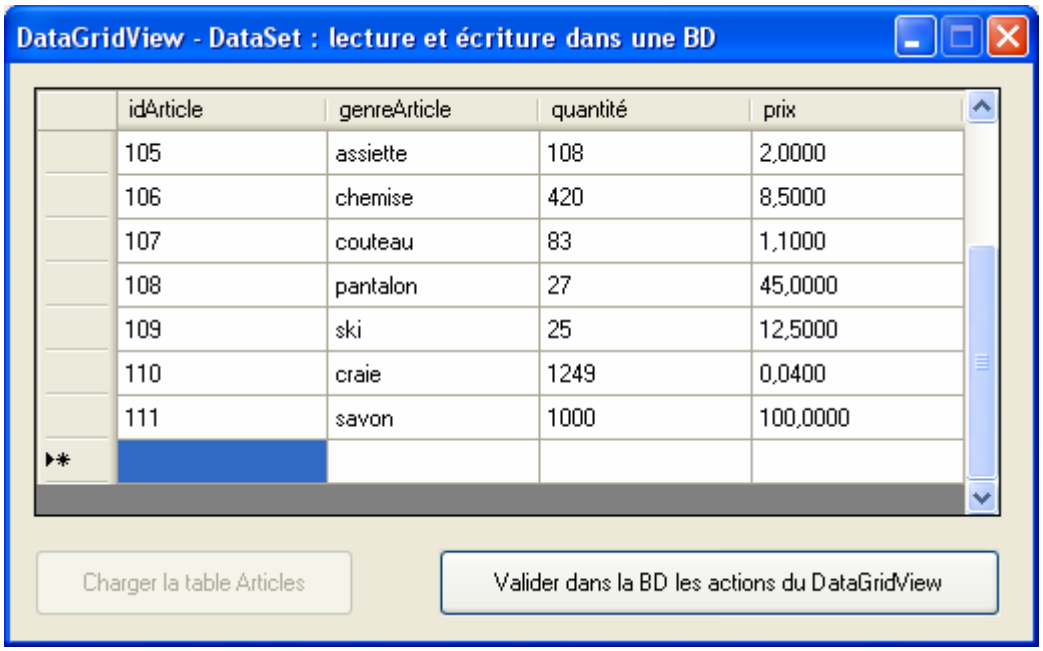
- un DataGridView nommé dataGridView1 affichant et autorisant les modifications de celulles,
- un bouton buttonLoad, de chargement de la table "Articles" à partir de la BD (à travers un DataSet lui-même connectée à la BD par un SqlDataAdapter),
- un bouton buttonValider, de sauvegarde et mise à jour dans la BD des modifications apportées dans le dataGridView1.




```
public partial class FormDatSetBD : Form
{
   public FormDatSetBD()
   {
      InitializeComponent();
   }
   string urlSqlServer = @"Data Source=(local);Initial Catalog=Boutique;" + "Integrated Security=SSPI;";
   DataSet undatSet = new DataSet();
   // objet de connexion SqlConnection :
   SqlConnection connexion;

   //objet de communication et d'échange de données
   SqlDataAdapter dataAdapteur = null;
```

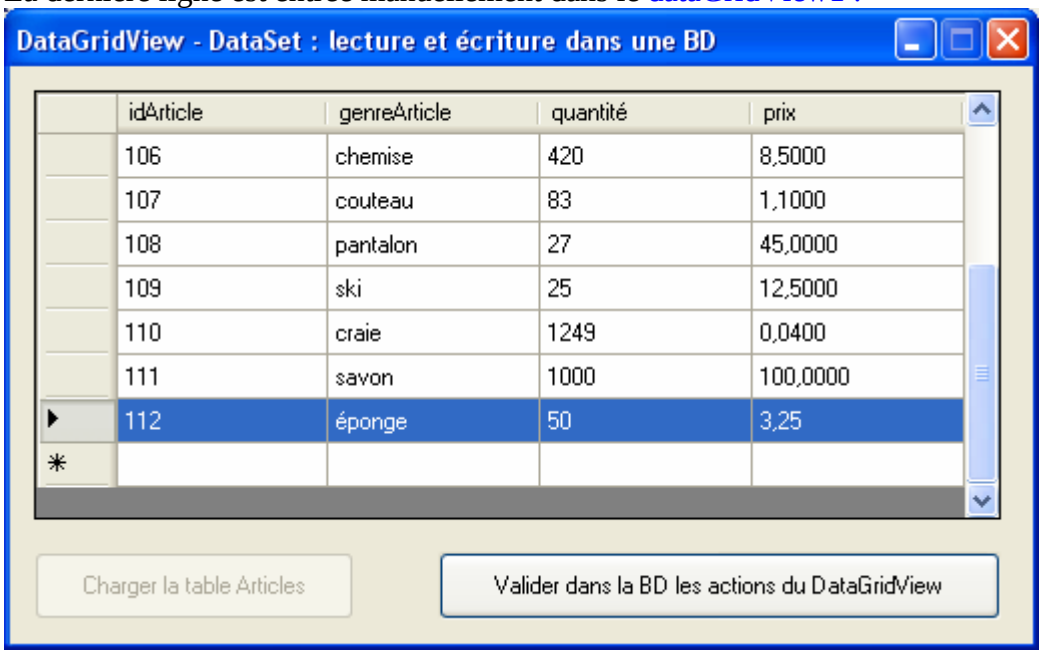
Chargement des données de la table "Articles" dans le dataGridView1 :

```
   private void buttonLoad_Click(object sender, EventArgs e)
   {
      // initialisation de l'objet de connexion SqlConnection :
      connexion = new SqlConnection(urlSqlServer);

      //objet de communication et d'échange branché sur la table Articles
      dataAdapteur = new SqlDataAdapter("Select * From Articles", connexion);
      // conseillé par Microsoft pour accéder à l'information de clef primaire
      dataAdapteur.MissingSchemaAction = MissingSchemaAction.AddWithKey;

      //Céation et remplissage d'une table 'LesArticles' dans le DataSet :
      dataAdapteur.Fill(undatSet, "LesArticles");

      //liaison de données de la table "LesArticles" avec le dataGridView1 :
      dataGridView1.DataSource = undatSet;
      dataGridView1.DataMember = undatSet.Tables[0].TableName;

      buttonLoad.Enabled = false;
      buttonValider.Enabled = true;
   }
```



Modification par envoi automatique d'une commande Transact-SQL :

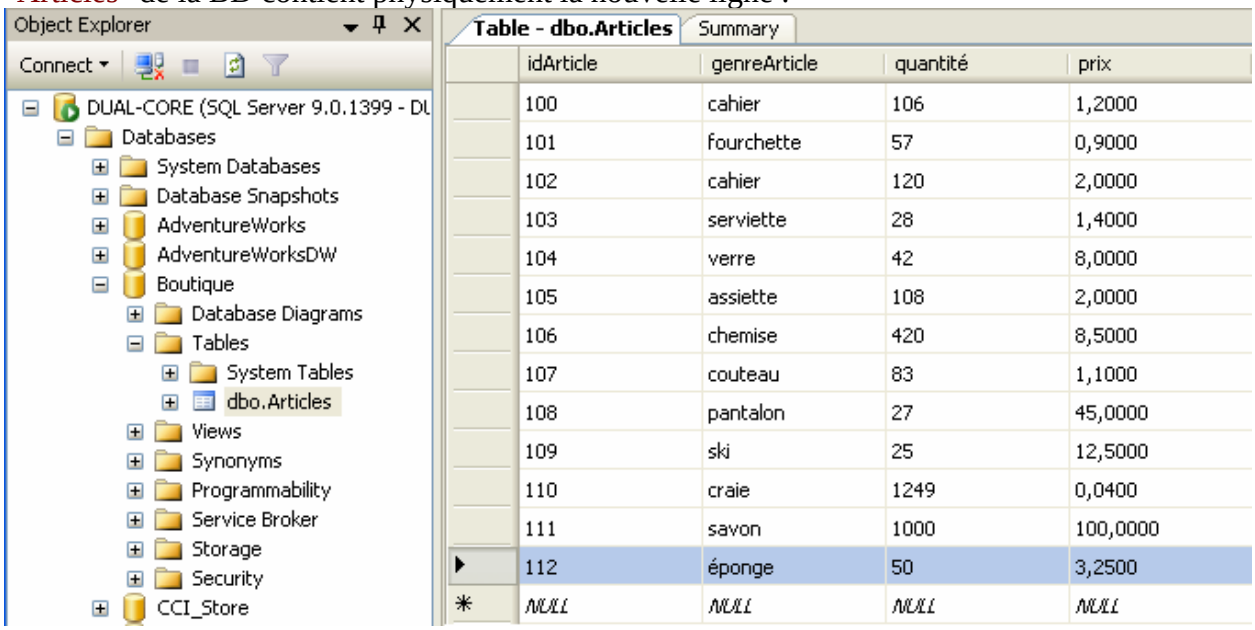
La dernière ligne est entrée manuellement dans le dataGridView1 :



```
//ce bouton valide les données dans la BD

private void buttonValider_Click(object sender, EventArgs e)
{
        // construction et lancement de la commande Transact-SQL insert, update ou delete:
        SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapteur);
        dataAdapteur.Update(undatSet, undatSet.Tables[0].TableName);
        /* validation effective des changements apportés (mettre après la commande Transact-SQL)
```

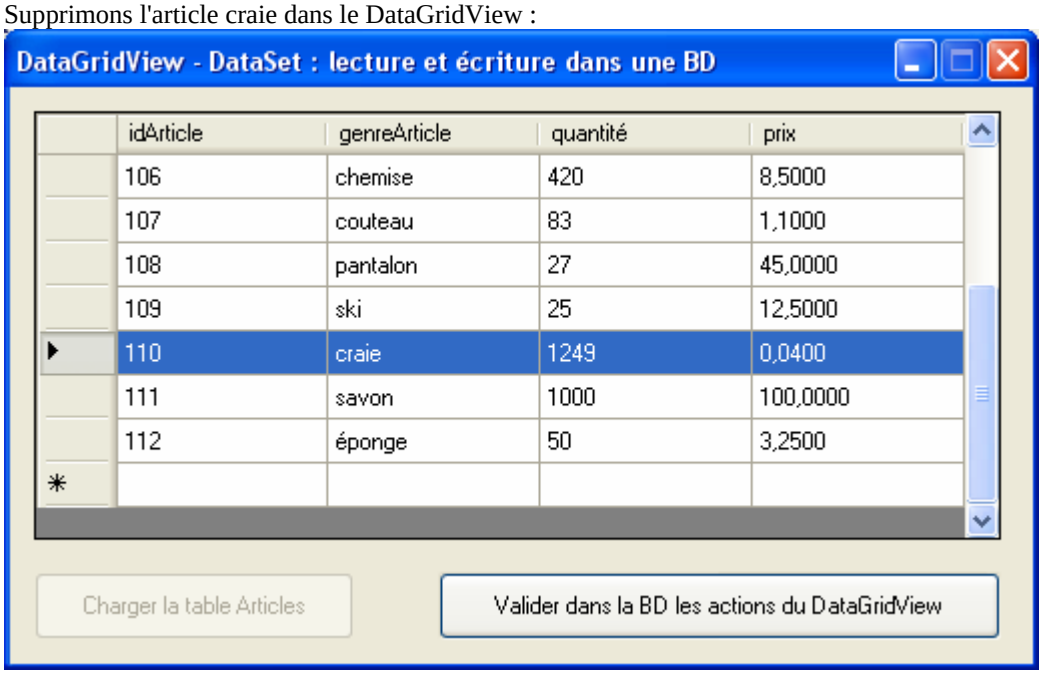
```
        * undatSet.AcceptChanges(); est automatiquement lancé par la méthode Update du dataAdapteur.
        * */
        buttonLoad.Enabled = true;
}
```

La méthode Update lance une commande SQL de type INSERT, après cette validation, la table "Articles" de la BD contient physiquement la nouvelle ligne :



idArticle	genreArticle	quantité	prix
100	cahier	106	1,2000
101	fourchette	57	0,9000
102	cahier	120	2,0000
103	serviette	28	1,4000
104	verre	42	8,0000
105	assiette	108	2,0000
106	chemise	420	8,5000
107	couteau	83	1,1000
108	pantalon	27	45,0000
109	ski	25	12,5000
110	craie	1249	0,0400
111	savon	1000	100,0000
112	éponge	50	3,2500
NULL	NULL	NULL	NULL

Supprimons l'article craie dans le DataGridView :



idArticle	genreArticle	quantité	prix
106	chemise	420	8,5000
107	couteau	83	1,1000
108	pantalon	27	45,0000
109	ski	25	12,5000
110	craie	1249	0,0400
111	savon	1000	100,0000
112	éponge	50	3,2500

Ensuite nous cliquons sur le bouton Valider, la méthode Update lance alors une commande SQL du type DELETE :

```
SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapteur);
dataAdapteur.Update(undatSet, undatSet.Tables[0].TableName);
```

Voici le nouvel état de la BD après cette validation :

Table - dbo.Articles | Summary

idArticle	genreArticle	quantité	prix
100	cahier	106	1,2000
101	fourchette	57	0,9000
102	cahier	120	2,0000
103	serviette	28	1,4000
104	verre	42	8,0000
105	assiette	108	2,0000
106	chemise	420	8,5000
107	couteau	83	1,1000
108	pantalon	27	45,0000
109	ski	25	12,5000
111	savon	1000	100,0000
112	éponge	50	3,2500
NULL	NULL	NULL	NULL

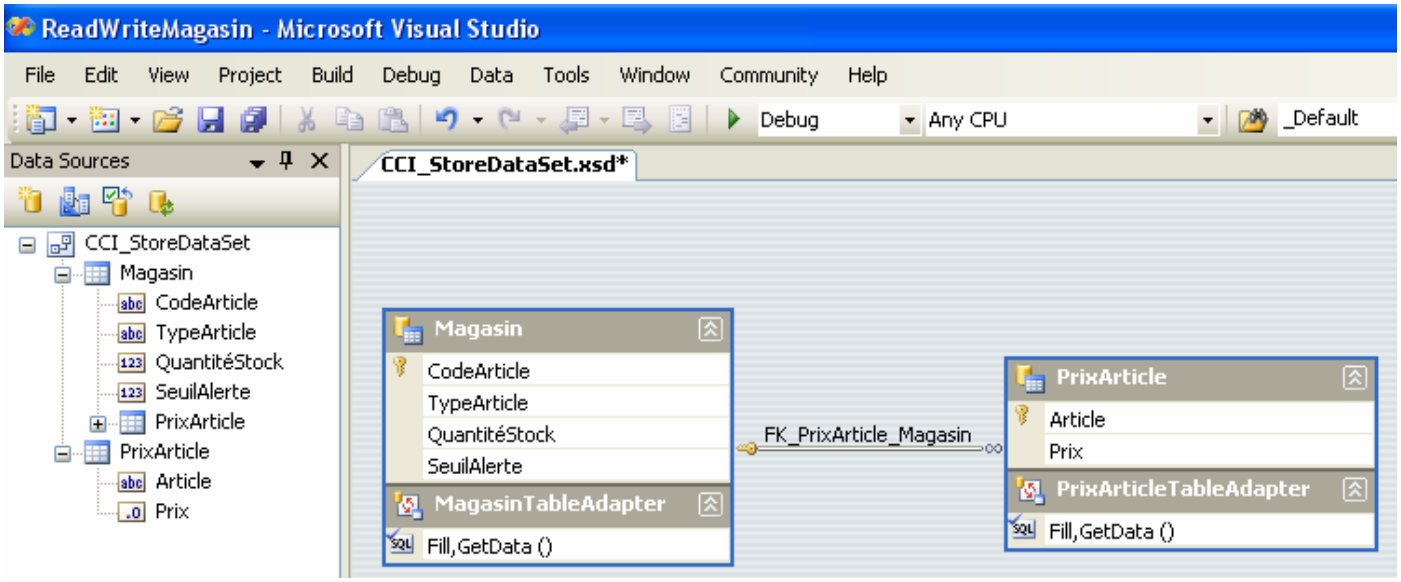
La ligne 110, craie, … a bien été effacée de la BD.

En conclusion :

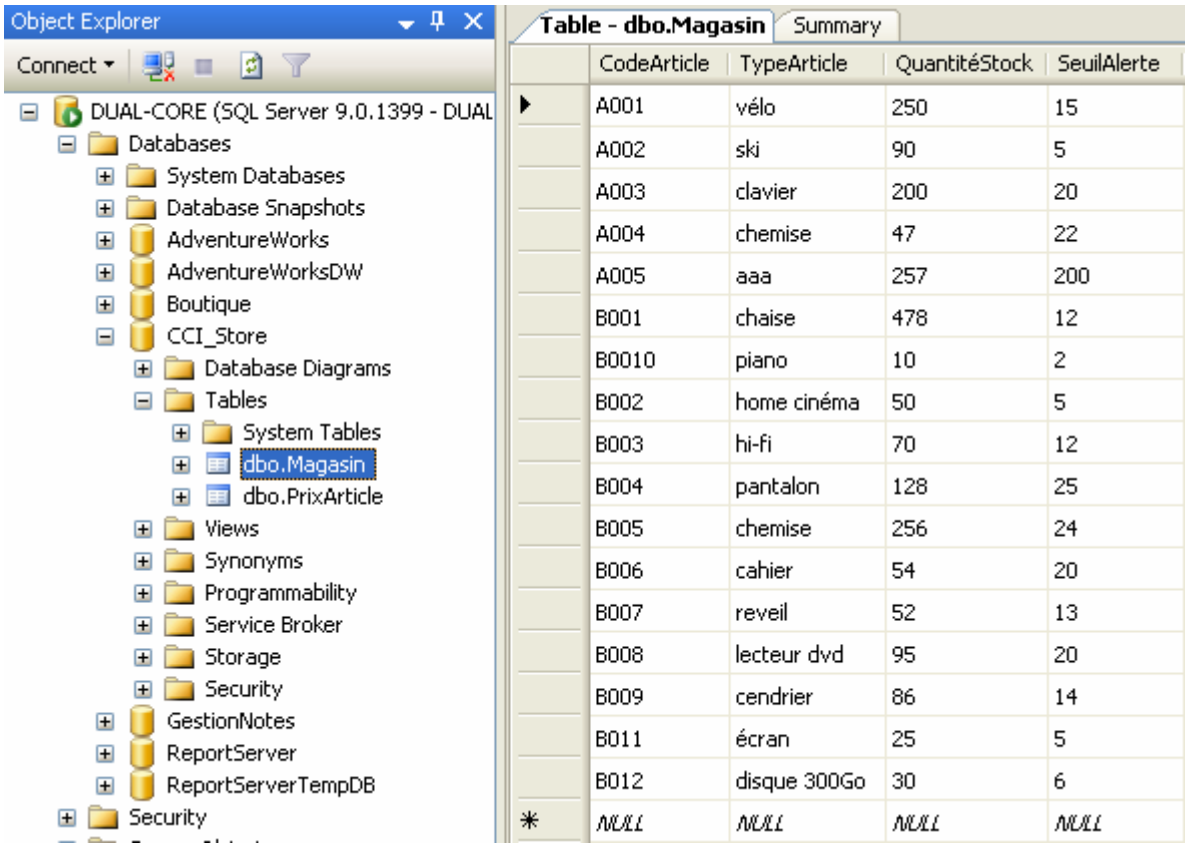
> Toute modification (insertion d'une ou plusieurs lignes, effacement d'une ou plusieurs lignes, changement des données incluses dans ou plusieurs cellules) est notifiée par le DataGridView au DataSet qui l'envoi au SQLDataAdapter qui lui-même génère les commandes Transact-SQL adéquates vers la BD afin que les mises à jours soient effectives.

Exercice : Gestion simplifiée d'un petit magasin

Soit une BD SQL serveur 2005 de gestion de stock nommée "CCI_Store.mdf" et comportant 2 tables liées entre elles :



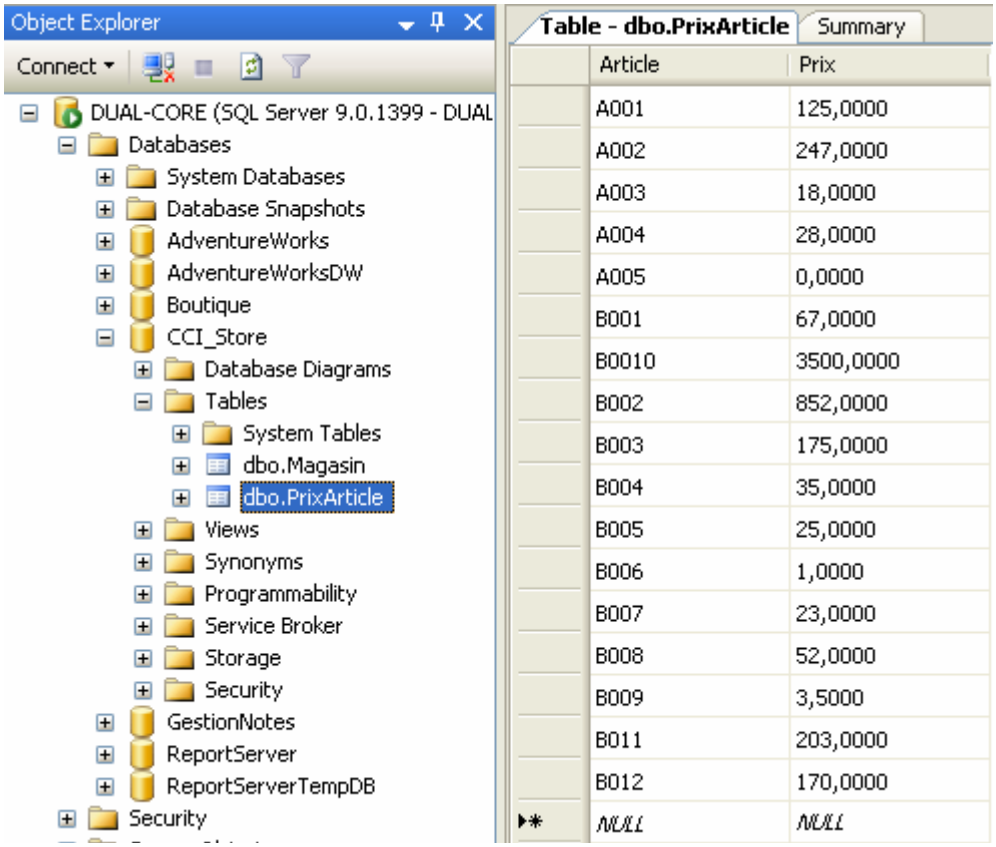
La table « Magasin » :



CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte
A001	vélo	250	15
A002	ski	90	5
A003	clavier	200	20
A004	chemise	47	22
A005	aaa	257	200
B001	chaise	478	12
B0010	piano	10	2
B002	home cinéma	50	5
B003	hi-fi	70	12
B004	pantalon	128	25
B005	chemise	256	24
B006	cahier	54	20
B007	reveil	52	13
B008	lecteur dvd	95	20
B009	cendrier	86	14
B011	écran	25	5
B012	disque 300Go	30	6
NULL	NULL	NULL	NULL

La colonne CodeArticle de la table Magasin est liée à la colonne Article de la table PrixArticle, c'est une clef ètrangère de la table Magasin qui réfère à la table PrixArticle

La table « PrixArticle » :



Programmation en « mode déconnecté » :

Implémentons en C#, les actions suivantes sur la BD "CCI_Store.mdf" :

1°) Le contenu de n'importe laquelle des cellules peut être modifié par l'utilisateur.
2°) Dès que le contenu d'une cellule est modifié la BD est mise-à-jour immédiatement.
3°) Dès que l'on clique dans une cellule, le contenu de la table est rafraîchi à partir de celui de la BD.
4°) L'application permet d'obtenir deux fichiers XML à partir de la BD :
    4.1°) La totalité de la BD au format XML.
    4.2°) Le schéma de la BD au format XSD.

On utilise un DataSet.

L'IHM de la gestion de stock affiche les deux tables et du XML obtenu à partir de la BD :

Gestion de stock

Charger les tables | Effacer les tables | Convertir la BD au format XML

Table : Magasin

CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte
A001	vélo	250	15
A002	ski	90	5
A003	ballon	200	20
B001	téléviseur	85	10
B002	home cinéma	50	5
B003	hi-fi	70	12
B004	lecteur dvd	100	20
B005	caméscope	50	5

Table : PrixArticle

Article	Prix
A001	125,0000
A002	247,0000
A003	18,0000
B001	256,0000
B002	852,0000
B003	175,0000
B004	198,0000
B005	720,0000

Schéma XML du DataSet | Contenu de la BD au format XML

```
<?xml version="1.0" encoding="utf-16"?>
<xs:schema id="NewDataSetCCI" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSetCCI" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="magasin">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CodeArticle">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:maxLength value="10" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
```

Une solution

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

namespace ReadWriteMagasin
{
   public partial class FStock : Form
   {
      private System.Data.DataSet dataSetStoreCCI;
      private string urlSqlServer = null;
      private SqlConnection connexion = null;
      private SqlDataAdapter magasinSqlAdapter = null;
      private SqlDataAdapter prixSqlAdapter = null;

      public FStock()
      {
         InitializeComponent();
         dataSetStoreCCI = new DataSet();//new CCI_StoreDataSet();
         dataSetStoreCCI.DataSetName = "NewDataSetCCI";
      }

      private DataTable loadTable(string nomTable, out SqlDataAdapter dataAdapteur)
      {
         DataTable table = new DataTable();
         //objet de communication et d'échange de données entre DataTable et la source de données (ici la BD)
         dataAdapteur = new SqlDataAdapter("Select * From " + nomTable, connexion);
         dataAdapteur.MissingSchemaAction = MissingSchemaAction.AddWithKey;

         //Remplissage de la table 'nomTable' :
         dataAdapteur.Fill(table);
         table.TableName = nomTable;
         return table;
      }

      private void displayTableMagasin()
      {
         //Remplissage du DataTable avec la table Magasin :
         DataTable table = loadTable("magasin", out magasinSqlAdapter);
         magasinSqlAdapter.Fill(table);

         //visualisation de la table "magasin" dans le dataGridViewMagasin :
         dataGridViewMagasin.DataSource = table;
         dataSetStoreCCI.Merge(table);
         dataSetStoreCCI.WriteXml("dsStoreMagasin.xml");
      }
```

```
        private void displayTablePrixArticle()
        {
            //Remplissage du DataTable avec la table PrixArticle :
            DataTable table = loadTable("PrixArticle", out prixSqlAdapter);
            prixSqlAdapter.Fill(table);

            //visualisation de la table "PrixArticle" dans le dataGridViewPrix :
            dataGridViewPrix.DataSource = table;
            dataSetStoreCCI.Merge(table);
            dataSetStoreCCI.WriteXml("dsStorePrix.xml");
        }

        private void buttonCharger_Click(object sender, EventArgs e)
        {
            try
            {
//--plusieurs chaînes de chemin de connexion possibles équivalents pour SQL server 2005  :
//-- (le serveur de l'auteur se nomme DUAL-CORE et se trouve à l'adresse réseau 10.5.8.1) :
//string urlSqlServer = @"Data Source=(local);Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
//string urlSqlServer = @"Data Source=localhost;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
//string urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
//string urlSqlServer = @"Data Source=10.5.8.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
//string urlSqlServer = @"Data Source=DUAL-CORE;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
                urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";

                // objet de connexion SqlConnection :
                connexion = new SqlConnection(urlSqlServer);

                displayTableMagasin();
                displayTablePrixArticle();

                // visualisation du schéma XML du dataSetStoreCCI
                toolStripButtonXML.Enabled = true;
                richTextBoxSchemaXML.Text = dataSetStoreCCI.GetXmlSchema();

                //par sécurité : validation effective des changements apportés
                dataSetStoreCCI.AcceptChanges();
            }
            catch (SqlException ex)
            {
                MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
            }
            finally
            {
                if (connexion != null)
                    connexion.Close();
            }
        }

        private void buttonSaveXML_Click(object sender, EventArgs e)
        {
            string FileName = "storeCCI.xml";
            dataSetStoreCCI.WriteXml(FileName, XmlWriteMode.IgnoreSchema);
            dataSetStoreCCI.WriteXmlSchema(FileName.Replace(".xml", ".xsl"));
            richTextBoxAllXML.Text = dataSetStoreCCI.GetXml();
        }

        private void buttonEffacer_Click(object sender, EventArgs e)
        {
            dataGridViewMagasin.DataSource = null;
```

```
        dataGridViewPrix.DataSource = null;
        toolStripButtonXML.Enabled = false;
    }

    /* -----   modifications de données dans la base   ----- */

    private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
    {  // modifications dans la table Magasin (sauf nouvelle ligne)
      if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count) //si pas nouvelle ligne
      {
        textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnName;
        textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex]=
                                                    dataGridViewMagasin[e.ColumnIndex,e.RowIndex].Value;

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
        magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

        // visualiser la commande Transact-SQL:
        Console.WriteLine(builder.GetUpdateCommand().CommandText);

        //par sécurité : validation effective des changements apportés
        dataSetStoreCCI.AcceptChanges();
      }
    }

    private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
    {
      if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count)
      {
        textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
        textBoxValColPrix.Text = Convert.ToString(dataGridViewPrix.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] =
                                                    dataGridViewPrix[e.ColumnIndex, e.RowIndex].Value;

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlAdapter);
        prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

        // visualiser la commande Transact-SQL:
        Console.WriteLine(builder.GetUpdateCommand().CommandText);

        //par sécurité : validation effective des changements apportés
        dataSetStoreCCI.AcceptChanges();
      }
    }

    private void dataGridViewMagasin_CellClick(object sender, DataGridViewCellEventArgs e)
    {
        // rafraîchissement des données présentes dans la base :
        displayTableMagasin();
        if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count) //si pas nouvelle ligne
         if (e.RowIndex >= 0 & e.ColumnIndex >= 0)  //car si ColumnIndex=-1, RowIndex=-1 on click alors dans les en-
têtes
          dataGridViewMagasin.CurrentCell = dataGridViewMagasin.Rows[e.RowIndex].Cells[e.ColumnIndex];
```

```
        }

        private void dataGridViewPrix_CellClick(object sender, DataGridViewCellEventArgs e)
        {
            // rafraîchissement des données présentes dans la base :
            displayTablePrixArticle();
            if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count) //si pas nouvelle ligne
             if (e.RowIndex >= 0 & e.ColumnIndex >= 0) //car si ColumnIndex=-1, RowIndex=-1 on click alors dans les en-
têtes
                dataGridViewPrix.CurrentCell = dataGridViewPrix.Rows[e.RowIndex].Cells[e.ColumnIndex];
        }
    }
}
```

Lors de l'exécution du programme précédent

Tentons de changer la référence B004 en référence B008 déjà existante :

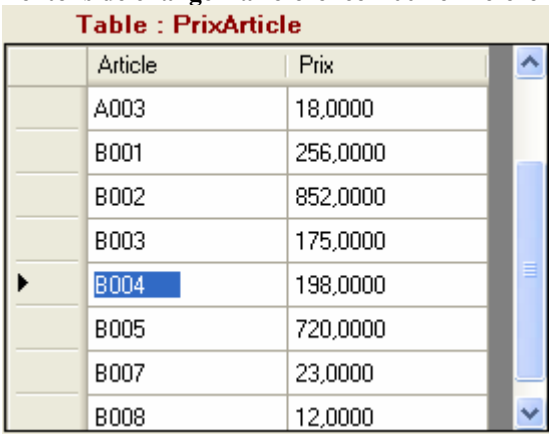


Table : PrixArticle

Article	Prix
A003	18,0000
B001	256,0000
B002	852,0000
B003	175,0000
B004	198,0000
B005	720,0000
B007	23,0000
B008	12,0000

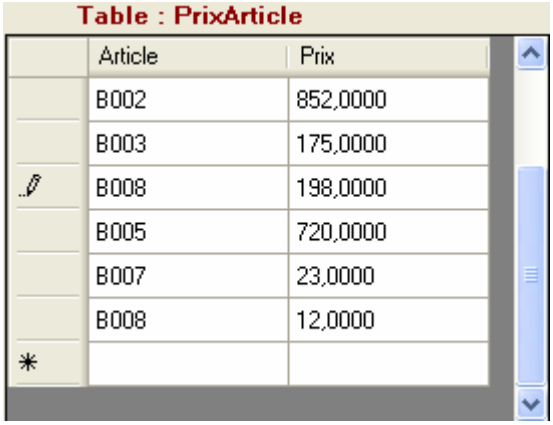


Table : PrixArticle

Article	Prix
B002	852,0000
B003	175,0000
B008	198,0000
B005	720,0000
B007	23,0000
B008	12,0000

.Net lance une exception du type **ConstraintException** :

Colunm "Article" est contrainte à l'unicité, la valeur B008 est déjà présente.

```
private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count)//si pas nouvelle ligne
    {
        textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
        textBoxValColPrix.Text = Convert.ToString(dataGridViewPrix.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewPrix[e.ColumnIndex, e.RowIndex].Value;

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new Sql
        prixSqlAdapter.Update(dataSetStoreC

        // visualiser la commande Transact-
        Console.WriteLine(builder.GetUpdate

        //par sécurité : validation effecti
        dataSetStoreCCI.AcceptChanges();
    }
}

private void dataGridViewMagasin_CellClick(
{
    // rafraîchissement des données présent
    displayTableMagasin();
```

ConstraintException was unhandled

Column 'Article' is constrained to be unique. Value 'B008' is already present.

Troubleshooting tips:

Relax or turn off constraints in your DataSet.
Clear datasets before loading them from view state.
Be sure you are not trying to assign a value to a primary key field whose primary key already exists.
Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...
Copy exception detail to the clipboard

Tentons de créer une nouvelle référence B009 à la place de la référence B004 déjà existante :

Table : PrixArticle

Article	Prix
A003	18,0000
B001	256,0000
B002	852,0000
B003	175,0000
B004	198,0000
B005	720,0000
B007	23,0000
B008	12,0000

Table : PrixArticle

Article	Prix
B002	852,0000
B003	175,0000
B009	198,0000
B005	720,0000
B007	23,0000
B008	12,0000

.Net lance une exception du type **SqlException** :

UPDATE créé un conflit avec la clef étrangère de nom FK_PrixArticle_Magasin…

```
private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e
{
    if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count)//si pas nouvelle ligne
    {
        textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnN
        textBoxValColPrix  textBoxNomColPrix {Text = "Article"}  GridViewPrix.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewPrix[e.Co

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlAdapter);
        prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

        // visualiser la commande Transact-SQL:
        Console.WriteLine(builder.Ge

        //par sécurité : validation
        dataSetStoreCCI.AcceptChange
    }
}

private void dataGridViewMagasin_Cel
{
    // rafraîchissement des données
    displayTableMagasin();
    if (e.RowIndex < dataSetStoreCCI
        if (e.RowIndex >= 0 & e.Colu
            dataGridViewMagasin.Curr
}
```

SqlException was unhandled

The UPDATE statement conflicted with the FOREIGN KEY constraint "FK_PrixArticle_Magasin". The conflict occurred in database "CCI_Store", table "dbo.Magasin", column 'CodeArticle'.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online…

Actions:

View Detail…

Copy exception detail to the clipboard

Si nous tentons de créer une nouvelle référence B009 à la place de la référence B004 déjà existante, mais cette fois-ci dans la table Magasin :

Table : Magasin

CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte
A003	ballon	200	20
B001	téléviseur	32	10
B002	home cinéma	50	5
B003	hi-fi	70	12
B004	lecteur dvd	100	20
B005	caméscope	50	5
B006	piano	56	8
B007	chapeau	250	40

.Net lance le même type **SqlException** :

UPDATE créé un conflit avec la clef étrangère de nom FK_PrixArticle_Magasin…

```
private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{  // modifications dans la table Magasin (sauf nouvelle ligne)
    if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count)//si pas nouvelle ligne
    {
        textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnNa
        textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewMagasin[e.Col

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
        magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

        // visualiser la commande Transact-SQL:
        Console.WriteL

        //par sécurité
        dataSetStoreC
    }
}

private void dataGridV                                          ewCellEventArgs e)
{
    if (e.RowIndex < d                                          lle ligne
    {
        textBoxNomColI                                          umnIndex].ColumnName;
        textBoxValColI                                          ntCell.Value);

        //modificatio
```

SqlException was unhandled

The UPDATE statement conflicted with the REFERENCE constraint "FK_PrixArticle_Magasin". The conflict occurred in database "CCI_Store", table "dbo.PrixArticle", column 'Article'.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

La colonne CodeArticle de la table Magasin est liée à la colonne Article de la table PrixArticle, c'est une clef ètrangère de la table Magasin qui réfère à la table PrixArticle, dans les deux derniers cas nous n'avons pas respecté

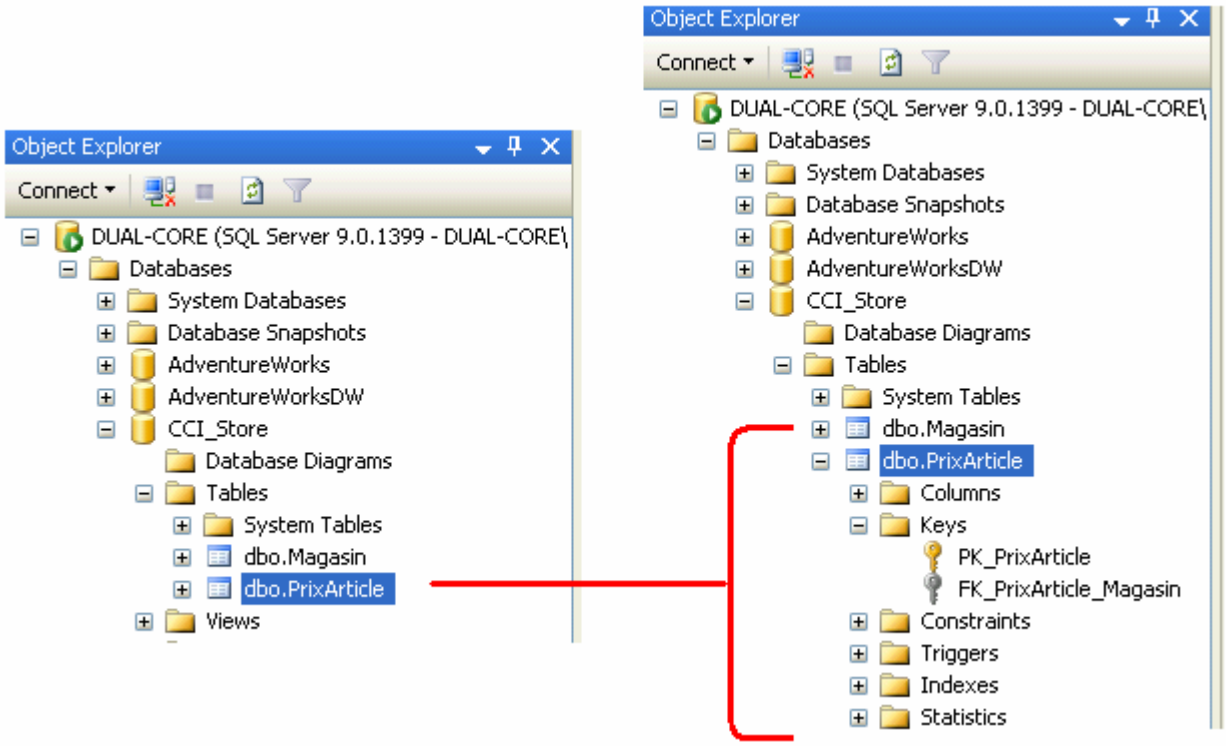
la règle d'intégrité référentielle (*toutes les valeurs d'une clef étrangère* doivent *se retrouver comme valeur de la clef primaire de la relation référée*) :

.Net renvoie les exceptions associées aux problèmes d'intégrités.

Amélioration de la gestion du petit magasin :
clef étrangère et delete/update en cascade

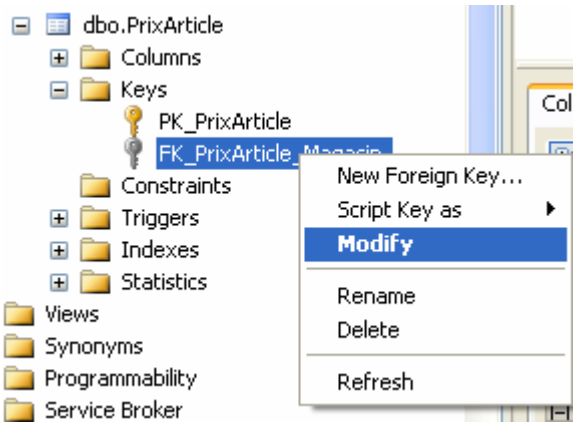
Soit les deux tables de l'exercice sur un petit magasin :

Microsoft SQL Server Management Studio permet de visualiser les deux clefs primaires de chacune des tables, et le fait que FK_PrixArticle_Magasin est une clef étrangère :

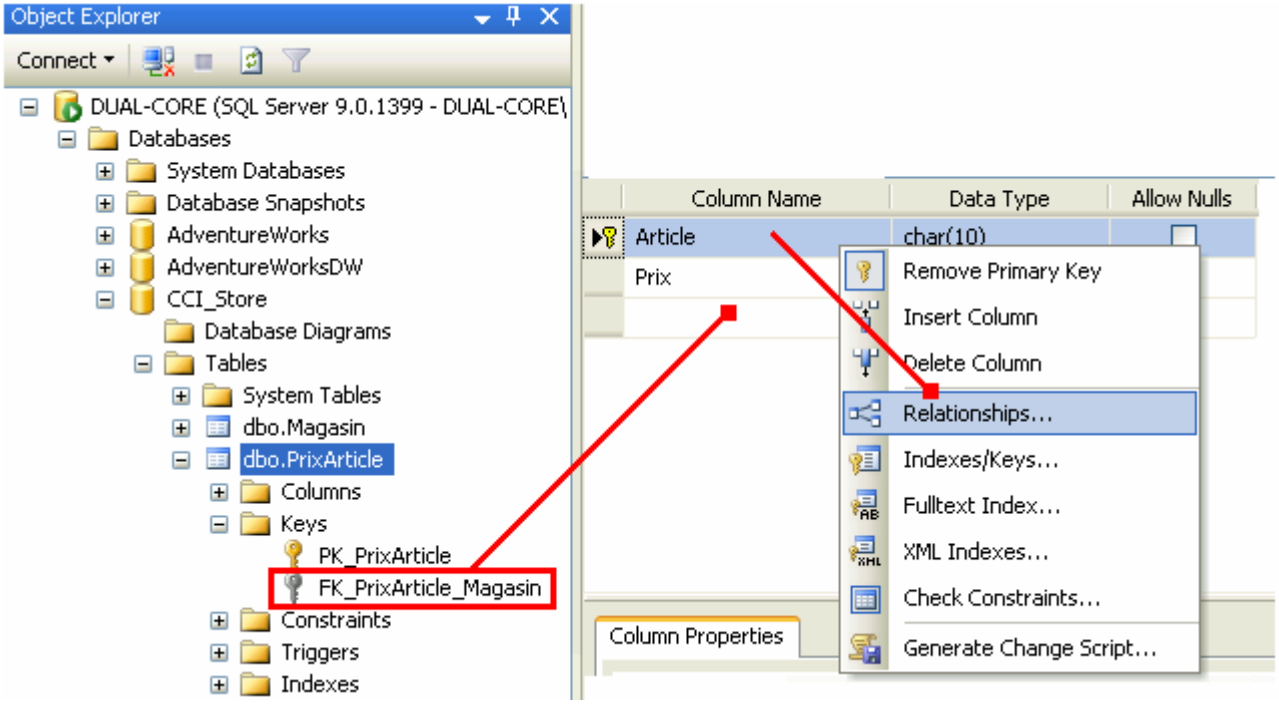


Il est possible avec SQL Server Management Studio et Visual Studio, de programmer visuellement la propagation de la mise à jour automatiquement de la clef étrangère aux relations aux quelles elle réfère (ici à la clef primaire de la table PrixArticle).

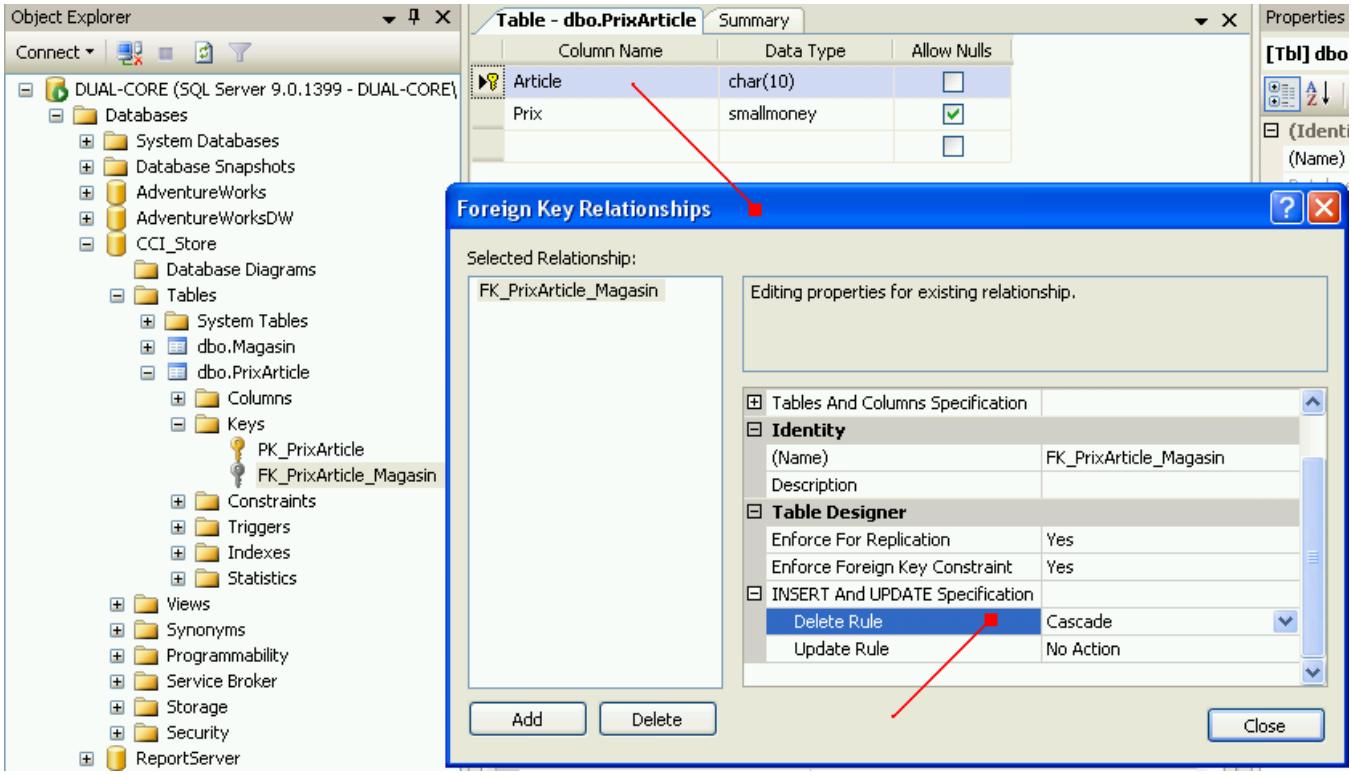
1°) Il faut soit sélectionner la clef étrangère FK_PrixArticle_Magasin et faire apparaître le menu pop-up associé par un click droit de souris et lancer la commande **Modify** :



2°) Soit effectuer la même opération (click droit de souris – commande Modify) sur le nom la table elle-même et ensuite faire apparaître à partir d'un click droit se souris sur la clef primaire la commande Relationship :

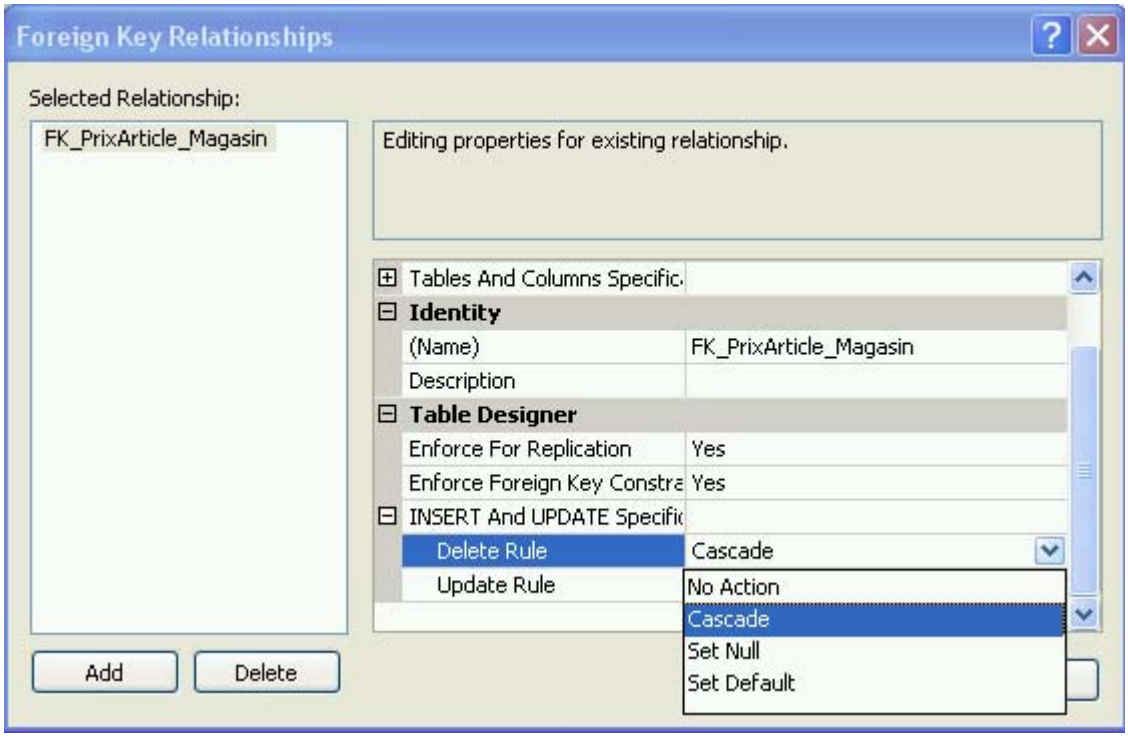


Ces deux opérations conduisent au même résultat : faire apparaître une fenêtre d'édition de propriétés pour les relations actuelles. Nous pouvons alors dans cet éditeur de propriétés, modifier les spécifications des commandes **UPDATE** et **DELETE**.



Pour chacune des commandes **UPDATE** et **DELETE** 4 choix sont possibles (**No Action** est le choix par défaut).

Nous choisissons de propager un DELETE dans la table Magasin à la table PrixArticle référée par la clef étrangère en utilisant le choix **Cascade** :



Dans le programme C#, la commande DELETE dans la table magasin :

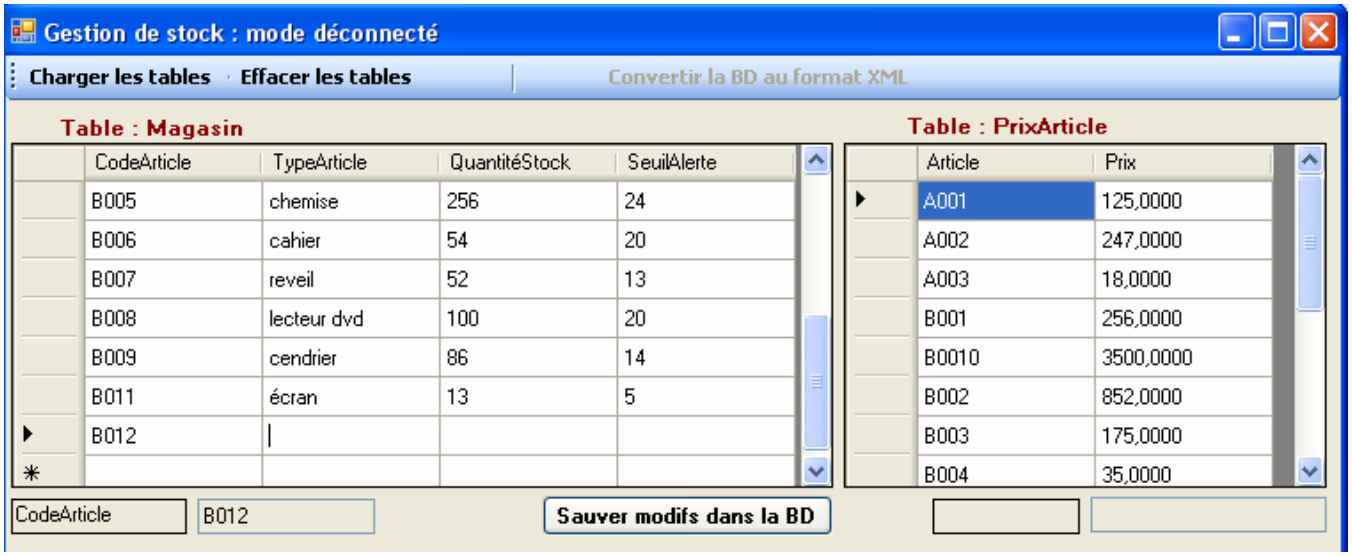
sendCommandTransact_SQL("supprimer", "magasin");

produit un DELETE sur la table Magasin et produit en cascade un DELETE sur la même ligne de la table prixArticle.

Implantation a effectuer

Sécurisez le programme de petit magasin comme suit :

- Chaque modification d'un ou plusieurs champs de la table Magasin est immédiatement et automatiquement validée dans la BD, sauf dans le cas de la création d'une nouvelle ligne où la validation des modifs sera proposée par l'activation d'un bouton.
- En empêchant toute modification du code Article dans la table PrixArticle à partir de l'IHM.
- En autorisant les modifications d'un ou plusieurs champs de la table Magasin, lorsqu'il s'agit du champ CodeArticle autoriser la propagation de la modification à la table PrixArticle.
- En permettant de supprimer avec confirmation, par un click droit de souris, une ligne entière de la table Magasin la propagation de la modification à la table PrixArticle.
- Lors de la création d'une nouvelle entrée dans la table Magasin, prévoir de ne pas mettre à jour le prix de l'article (qui est automatiquement mis à 0.00 €) tant que la ligne entière n'a pas été sauvegardée dans la BD.



Une solution de l'mplantation

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

namespace ModifMagasin
{
   public partial class FModifStock : Form
   {
      private System.Data.DataSet dataSetStoreCCI;
      private string urlSqlServer = null;
      private SqlConnection connexion = null;
      private SqlDataAdapter magasinSqlAdapter = null;
      private SqlDataAdapter prixSqlAdapter = null;
      //--pour SQL server Ed.Developper 2005 :
      //string urlSqlServer = @"Data Source=(local);Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
      //string urlSqlServer = @"Data Source=localhost;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
      //string urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
      //string urlSqlServer = @"Data Source=10.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
      //string urlSqlServer = @"Data Source=DUAL-CORE;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";

      public FModifStock()
      {
         InitializeComponent();
         dataSetStoreCCI = new DataSet();
         dataSetStoreCCI.DataSetName = "NewDataSetCCI";
      }


      private DataTable loadTable(string nomTable, out SqlDataAdapter dataAdapteur)
      {
         DataTable table = new DataTable();
         //objet de communication et d'échange de données entre DataTable et la source de données (ici la BD)
         dataAdapteur = new SqlDataAdapter("Select * From " + nomTable, connexion);
         dataAdapteur.MissingSchemaAction = MissingSchemaAction.AddWithKey;

         //Remplissage de la table 'nomTable' :
         dataAdapteur.Fill(table);
         table.TableName = nomTable;
         return table;
      }

      private void displayTableMagasin()
      {
         try
         {
            urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
            // objet de connexion SqlConnection :
            connexion = new SqlConnection(urlSqlServer);

            //Remplissage du DataTable avec la table Magasin :
            DataTable table = loadTable("magasin", out magasinSqlAdapter);
            magasinSqlAdapter.Fill(table);

            //visualisation de la table "magasin" dans le dataGridViewMagasin :
            dataGridViewMagasin.DataSource = table;
            dataSetStoreCCI.Merge(table);
            dataSetStoreCCI.WriteXml("dsStoreMagasin.xml");
         }
```

```
    catch (SqlException ex)
    {
        MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    finally
    {
        if (connexion != null)
            connexion.Close();
    }
}

private void displayTablePrixArticle()
{
    try
    {
        urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        // objet de connexion SqlConnection :
        connexion = new SqlConnection(urlSqlServer);
        //Remplissage du DataTable avec la table PrixArticle :
        DataTable table = loadTable("PrixArticle", out prixSqlAdapter);
        prixSqlAdapter.Fill(table);

        //visualisation de la table "PrixArticle" dans le dataGridViewPrix :
        dataGridViewPrix.DataSource = table;
        dataGridViewPrix.Columns[0].ReadOnly = true;
        dataSetStoreCCI.Merge(table);
        dataSetStoreCCI.WriteXml("dsStorePrix.xml");
    }
    catch (SqlException ex)
    {
        MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    finally
    {
        if (connexion != null)
            connexion.Close();
    }
}

private void buttonCharger_Click(object sender, EventArgs e)
{
    displayTableMagasin();
    displayTablePrixArticle();

    // visualisation du schéma XML du dataSetStoreCCI
    toolStripButtonXML.Enabled = true;
    richTextBoxSchemaXML.Text = dataSetStoreCCI.GetXmlSchema();
}

private void buttonSaveXML_Click(object sender, EventArgs e)
{
    string FileName = "storeCCI.xml";
    dataSetStoreCCI.WriteXml(FileName, XmlWriteMode.IgnoreSchema);
    dataSetStoreCCI.WriteXmlSchema(FileName.Replace(".xml", ".xsl"));
    richTextBoxAllXML.Text = dataSetStoreCCI.GetXml();
}

private void buttonEffacer_Click(object sender, EventArgs e)
{
    dataGridViewMagasin.DataSource = null;
    dataGridViewPrix.DataSource = null;
    toolStripButtonXML.Enabled = false;
}

private void sendCommandTransact_SQL(string idCommand, string nomTable)
{
    SqlDataAdapter SqlDataAdapteur;
```

```
        if (nomTable == "magasin")
            SqlDataAdapteur = magasinSqlAdapter;
        else
            SqlDataAdapteur = prixSqlAdapter;

        // construction et lancement de la commande Transact-SQL insert, update ou delete:
        SqlCommandBuilder builder = new SqlCommandBuilder(SqlDataAdapteur);
        SqlDataAdapteur.Update(dataSetStoreCCI, nomTable);

        // visualiser la commande Transact-SQL:
        switch (idCommand)
        {
            case "inserer": Console.WriteLine(builder.GetInsertCommand().CommandText); break;
            case "modifier": Console.WriteLine(builder.GetUpdateCommand().CommandText); break;
            case "supprimer": Console.WriteLine(builder.GetDeleteCommand().CommandText); break;
        }
    }

    /* -----   modifications de données dans la base   ----- */

    /*
     * Lorsque l'on entre une nouvelle ligne à la fin (nouvel article) dans
     * la table "magasin", il y a création d'une nouvelle ligne avec la même
     * clef article et un prix à 0,00€ dans la table "prixarticle".
     * */
    private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
    {
        // modifications dans la table Magasin
        textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnName;
        textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentCell.Value);
        if (e.RowIndex <= dataSetStoreCCI.Tables[0].Rows.Count - 1)// mode modification de données d'une ligne existante
        {
            //modification du dataSetStoreCCI :
            dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewMagasin[e.ColumnIndex,
e.RowIndex].Value;

            // construction et lancement de la commande Transact-SQL:
            sendCommandTransact_SQL("modifier", "magasin");
            // <=>
            //SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
            //magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

            // visualiser la commande Transact-SQL:
            //Console.WriteLine(builder.GetUpdateCommand().CommandText);

            //par sécurité : validation effective des changements apportés(mettre après la commande Transact-SQL)
            dataSetStoreCCI.AcceptChanges();
            if (e.ColumnIndex == 0)//on vient de changer un code Article, alors on réaffiche les données
            {
                dataSetStoreCCI = new DataSet();
                dataSetStoreCCI.DataSetName = "NewDataSetCCI";
                displayTableMagasin();
                displayTablePrixArticle();
            }
        }// sinon création d'une nouvelle ligne en fin de tableau
        else
        {
            //modification de la table magasin du dataSetStoreCCI :
            if (dataSetStoreCCI.Tables[0].Rows.Count < dataGridViewMagasin.RowCount)
            {
                // on crée la ligne  dans la table magasin
                DataRow Line = dataSetStoreCCI.Tables[0].NewRow();
                Line[e.ColumnIndex] = dataGridViewMagasin[e.ColumnIndex, e.RowIndex].Value;
                dataSetStoreCCI.Tables[0].Rows.Add(Line);
                // construction et lancement de la commande Transact-SQL:
                sendCommandTransact_SQL("modifier", "magasin");
```

```
            // on crée la ligne correspondante dans la table PrixArticle
            if (dataGridViewMagasin[e.ColumnIndex, e.RowIndex].Value != DBNull.Value)
            {
              DataRow newLine = dataSetStoreCCI.Tables[1].NewRow();
              newLine[0] = Line[0];
              newLine[1] = 0;
              dataSetStoreCCI.Tables[1].Rows.Add(newLine);
              // construction et lancement de la commande Transact-SQL:
              sendCommandTransact_SQL("modifier", "PrixArticle");
              displayTablePrixArticle();
            }
          }
          buttonSave.Enabled = true;
          dataGridViewPrix.ReadOnly = true;// aucune opération acceptée tant que la validation n'a pas eu lieu
        }
      }

      private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
      {
        textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
        textBoxValColPrix.Text = Convert.ToString(dataGridViewPrix.CurrentCell.Value);
        if (e.ColumnIndex != 0)
          if ((e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count - 1)
            | e.RowIndex == dataSetStoreCCI.Tables[1].Rows.Count - 1)// mode modification de données d'une ligne existante
          {
            // modification du dataSetStoreCCI :
            dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewPrix[e.ColumnIndex,
e.RowIndex].Value;

            // construction et lancement de la commande Transact-SQL:
            sendCommandTransact_SQL("modifier", "PrixArticle");
            // <=>
            //SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlAdapter);
            //prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

            // visualiser la commande Transact-SQL:
            //Console.WriteLine(builder.GetUpdateCommand().CommandText);

            // validation effective des changements apportés (mettre après la commande Transact-SQL)
            dataSetStoreCCI.AcceptChanges();
            displayTablePrixArticle();

          }// sinon création d'une nouvelle ligne interdite
          else
          {
            //rien pour l'instant....
          }
      }

      private void dataGridViewMagasin_MouseDown(object sender, MouseEventArgs e)
      {// on supprime une ligne sélectionnée par click droit de souris sur cette ligne :
        int nbrRow = dataGridViewMagasin.SelectedRows.Count;
        int numRow = dataGridViewMagasin.CurrentCell.RowIndex;
        if (nbrRow != 0 & e.Button == MouseButtons.Right)
        {
          if (MessageBox.Show(this, "Confirmez-vous la suppression de cet article de la Base ?", "Suppression de la ligne
CodeArticle : "
            + dataGridViewPrix.Rows[numRow].Cells[0].Value + "demandée.",
            MessageBoxButtons.YesNo) == DialogResult.Yes)
          {
            //dataSetStoreCCI.Tables[0].Rows.RemoveAt(numRow); // ne produit pas de DELETE du magasinSqlAdapter
            dataSetStoreCCI.Tables[0].Rows[numRow].Delete();
            dataGridViewMagasin.DataSource = dataSetStoreCCI.Tables[0];
            //dataGridViewMagasin.Update();

            // construction et lancement de la commande Transact-SQL:
            /* DELETE-UPDATE en cascade dans la définition de la clef étrangère "FK_PrixArticle_Magasin"
```

```
            * dans la table PrixArticle.(cf ClefEtrangereCascade.doc)
            * */
            sendCommandTransact_SQL("supprimer", "magasin");
            // <=>
            //SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
            //magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

            // visualiser la commande Transact-SQL:
            //Console.WriteLine(builder.GetUpdateCommand().CommandText);

            // validation effective des changements apportés (mettre après la commande Transact-SQL)
            dataSetStoreCCI.AcceptChanges();
            dataSetStoreCCI = new DataSet();
            dataSetStoreCCI.DataSetName = "NewDataSetCCI";
            displayTableMagasin();
            displayTablePrixArticle();
        }
      }
    }

    private void buttonSave_Click(object sender, EventArgs e)
    {
      // construction et lancement de la commande Transact-SQL insert, update ou delete:
      SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
      magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

      builder = new SqlCommandBuilder(prixSqlAdapter);
      prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

      // validation effective des changements apportés (mettre après la commande Transact-SQL)
      dataSetStoreCCI.AcceptChanges();
      displayTableMagasin();
      displayTablePrixArticle();
      buttonSave.Enabled = false;
      dataGridViewPrix.ReadOnly = false;
    }

    private void FStock_FormClosing(object sender, FormClosingEventArgs e)
    {
      //buttonSave_Click(sender, new EventArgs());
    }
  }
}
```

Programmation web avec ASP.Net

- **Page web écrite uniquement avec des contrôles HTML simples**
- **Page web écrite uniquement avec des contrôles serveur**
- **Communication entre pages web**
- **Page web écrite avec BD et GridView**
- **Cookie dans une page web**

Page web écrite avec des contrôles HTML

Plan général:

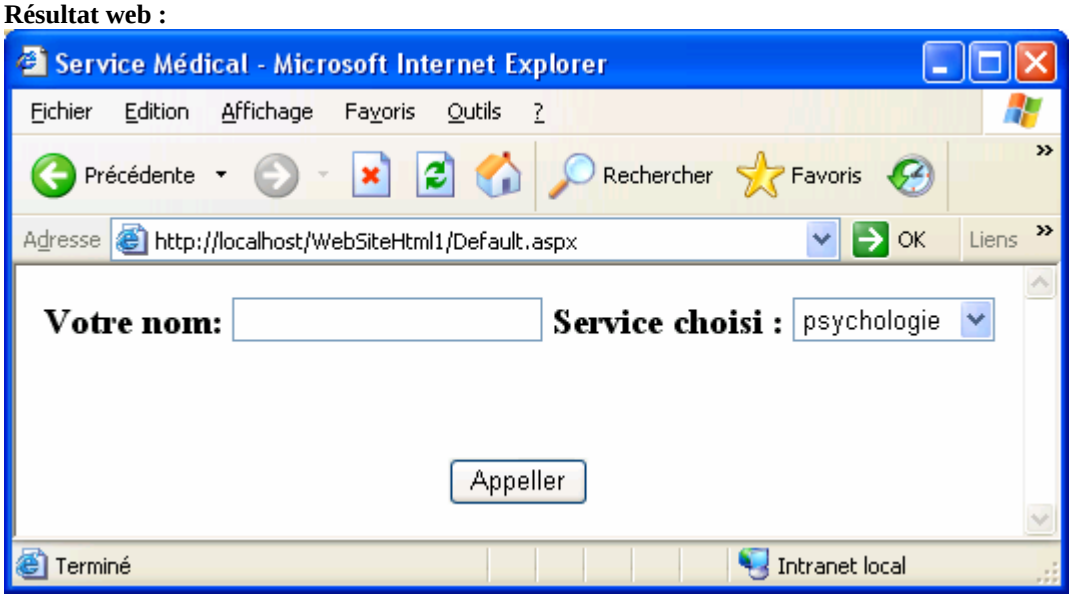
Page web écrite uniquement avec des contrôles HTML simples

1°) Page HTML classique
2°) Ajouter du code élémentaire C# <% … %> dans la page HTML
3°) Réponse à une requête post back de la part du client
4°) Utiliser Visual Studio pour générer du HTML
5°) Utiliser Visual Studio pour générer des appels au javascript
6°) Utilisez du javascript pour onmouseover et onmousemove

1°) Page HTML classique :

```
<html>
   <head>
      <Title>Service Médical</Title>
   </head>
   <body>
       <center>
       <form action="Default.aspx">
           <h3> Votre nom: <input id="Nom" type=text>

           Service choisi :  <select id="Categorie" size=1>
                             <option>psychologie</option>
                             <option>pédiatrie</option>
                             <option>dermatologie</option>
                         </select>
           </h3>
           <br/>
           <br/>
           <input type=submit value="Appeler">
       </form>
       </center>
   </body>
</html>
```

Résultat web :

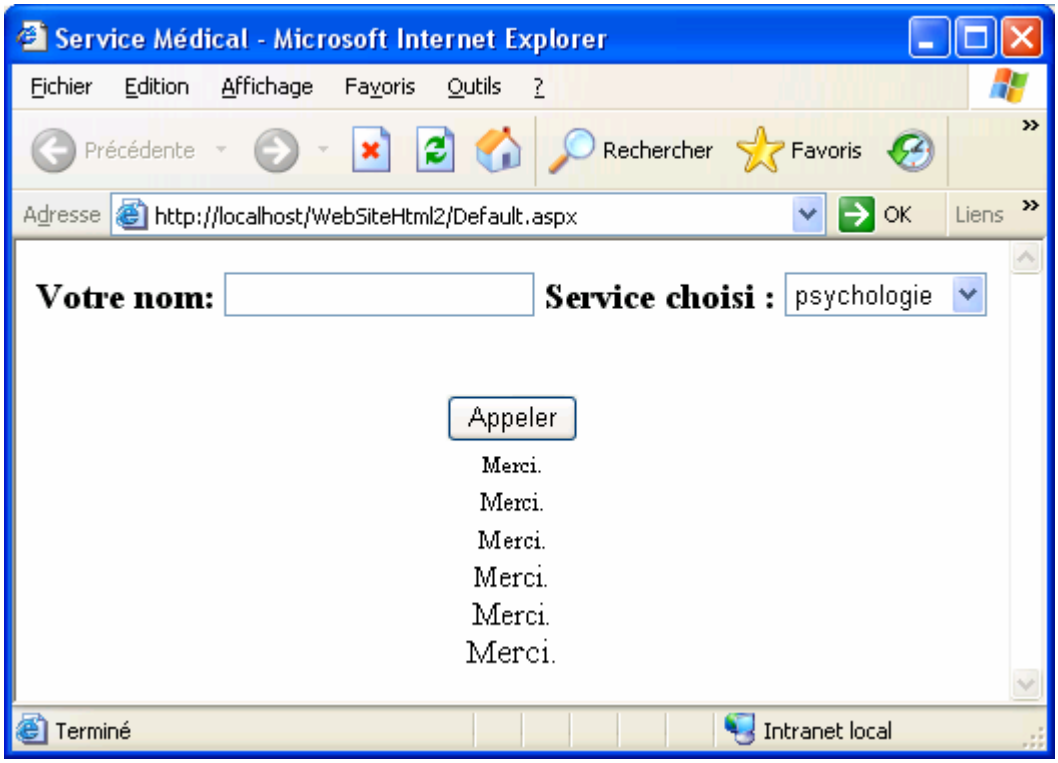


2°) Ajouter du code élémentaire C# <% … %> dans la page HTML

```
<%@ Page Language="C#" %>
<html>
   <head>
      <Title>Service Médical</Title>
   </head>
   <body>
       <center>
```

```
        <form action="Default.aspx">
            <h3> Votre nom: <input id="Nom" type=text>

            Service choisi :  <select id="Categorie"
size=1>
                                <option>psychologie</option>
                                <option>pédiatrie</option>
                                <option>dermatologie</option>
                            </select>
            </h3>
            <br/>
            <input type=submit value="Appeler">
            <br/>
            <% for (int i=8;i<14; i++) { %>
            <span style="font-size: <%=i%>pt">
Merci.</span>
            <br/>
            <% } %>
        </form>
        </center>
    </body>
</html>
```

Résultat web :

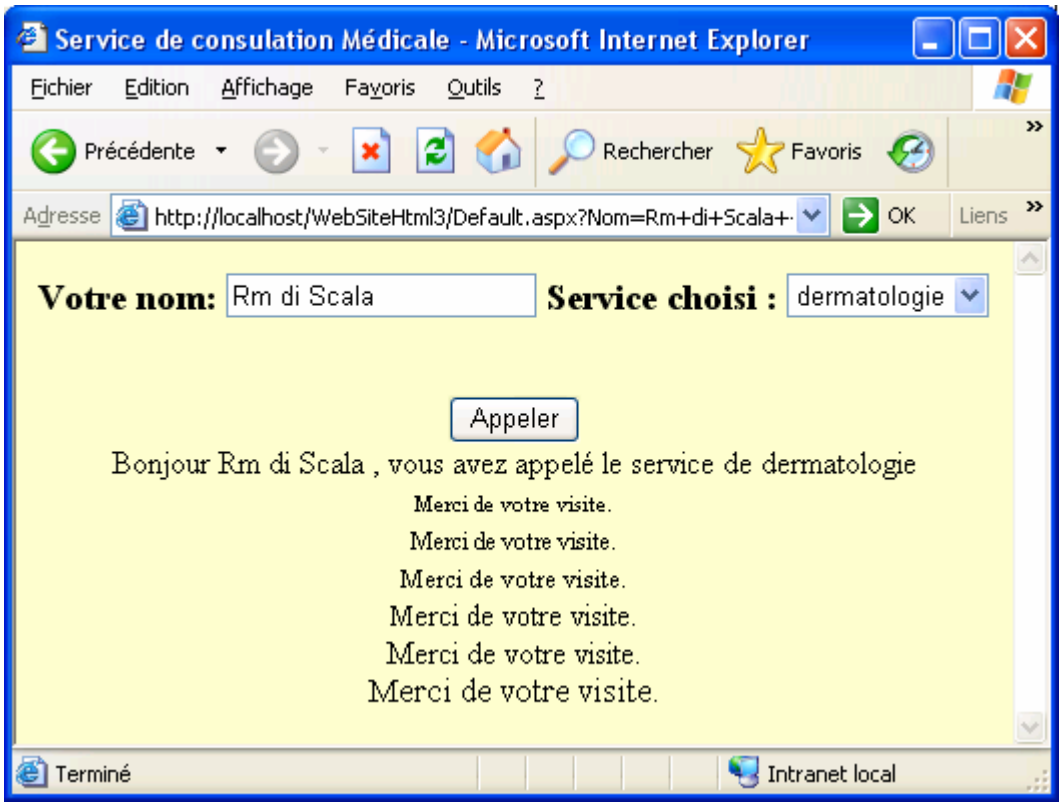


Important : le code inclus entre les blocs **<% ... %>** ne doit contenir **ni classe**, **ni méthode**, il est compilé effectivement sur la machine client (contrairement aux langages de scripts qui sont interprétés).

3°) Réponse à une requête post back de la part du client

```
<%@ Page Language="C#" %>
<html>
   <head>
      <Title>Service de consulation Médicale</Title>
   </head>
```

```
   <body style="background-color :#FFFFCC">
       <center>
       <form action="Default.aspx">
           <h3> Votre nom: <input name="Nom" type=text
            value="<%=HttpUtility.HtmlEncode(Request.QueryString["Nom"])%>
            ">
           Service choisi :  <select name="service" size=1>
           <%string[] strChoix={"psychologie","pédiatrie","dermatologie"};
             for (int i = 0; i < strChoix.Length; i++)
             {    %>
                 <option
                 <%if (Request.QueryString["service"]==strChoix[i])
                    Response.Write("selected");
                 %>
                 >
                 <%=strChoix[i] %>
                 </option>
             <% } %>
            </select>
           </h3>
           <br/>
           <input type=submit name="choix" value="Appeler">
           <% if (Request.QueryString["choix"]!=null) { %>
               <br/>
                Bonjour
                <%=HttpUtility.HtmlEncode(Request.QueryString["Nom"])%>,
                vous avez appelé le service de
                <%=HttpUtility.HtmlEncode(Request.QueryString["service"])%>
                 <br/>
                  <%for (int i=8;i<14; i++) { %>
                  <span style="font-size: <%=i%>pt"> Merci de votre
visite.</span>
                  <br/>
                  <%} %>
           <% } %>
       </form>
       </center>
   </body>
</html>
```



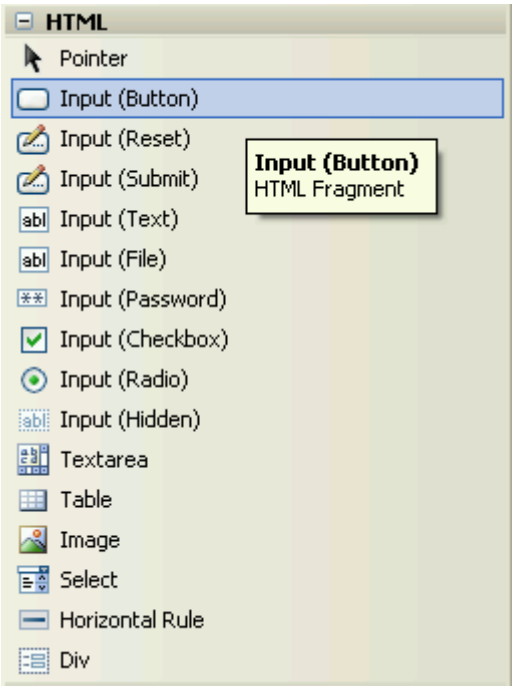
4°) Utiliser Visual Studio pour générer du HTML

L'environnement de programmation Visual Studio permet de créer visuellement votre interface HTML avec des entités visuelles générant immédiatement dans votre page source le code HTML correspondant.

La palette d'outils de Visual Studio propose 15 entités permettant de générer 15 types de balises de code HTML différentes. Ces contrôles HTML se trouvent dans le namespace System.Web.UI.HtmlControls, ils sont présents dans la palette des contrôles "toolbox" dans l'onglet HTML :

Premier exemple d'utilisation de HtmlControls : afficher un bouton "Input" dans une page web.

Onglet HTML de la Toolbox :



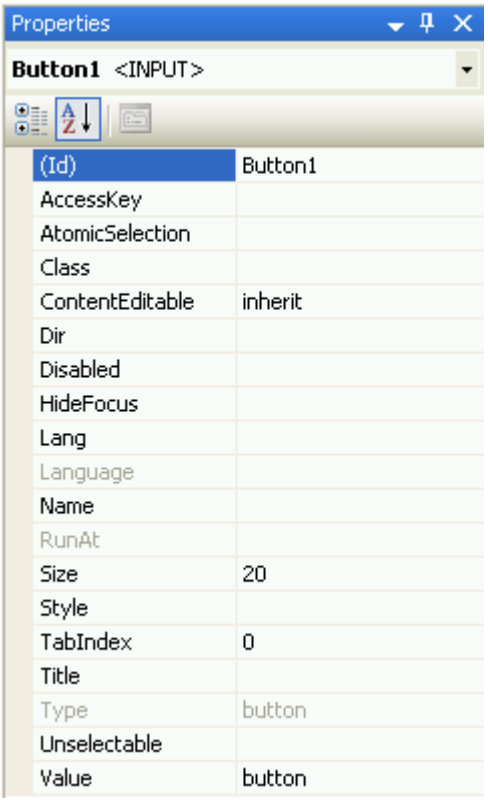
Le HtmlControls choisi dans la Toolbox est le *Input Button.*

Entité visuelle :

Code HTML engendré :

```
<input id="Button1" type="button" value="button" />
```

Propriétés paramétrables :

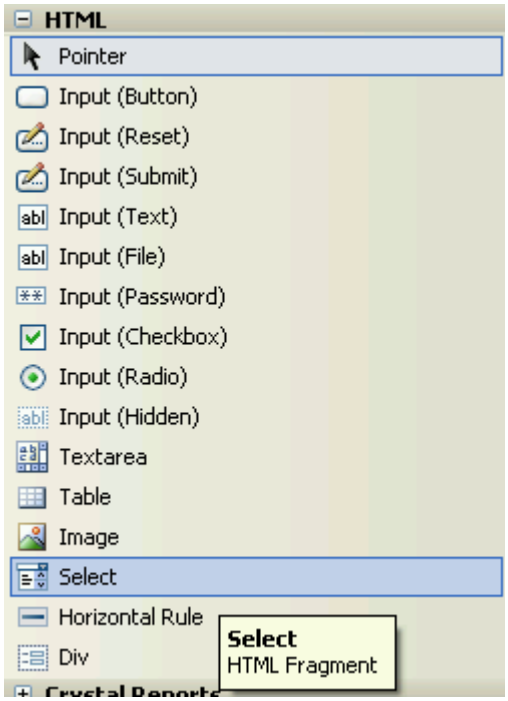


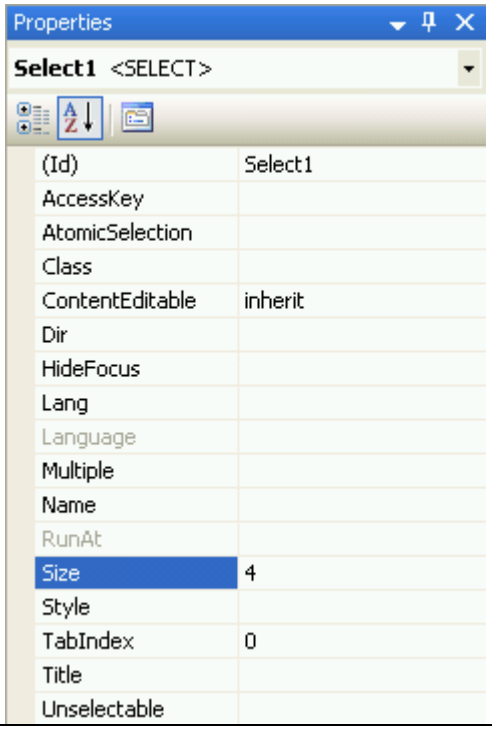
Le code source HTML obtenu est insérable dans une page :

```
<html>
<head>
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1">
    <div>
        <input id="Button1" type="button" value="button" />
    </div>
    </form>
</body>
</html>
```

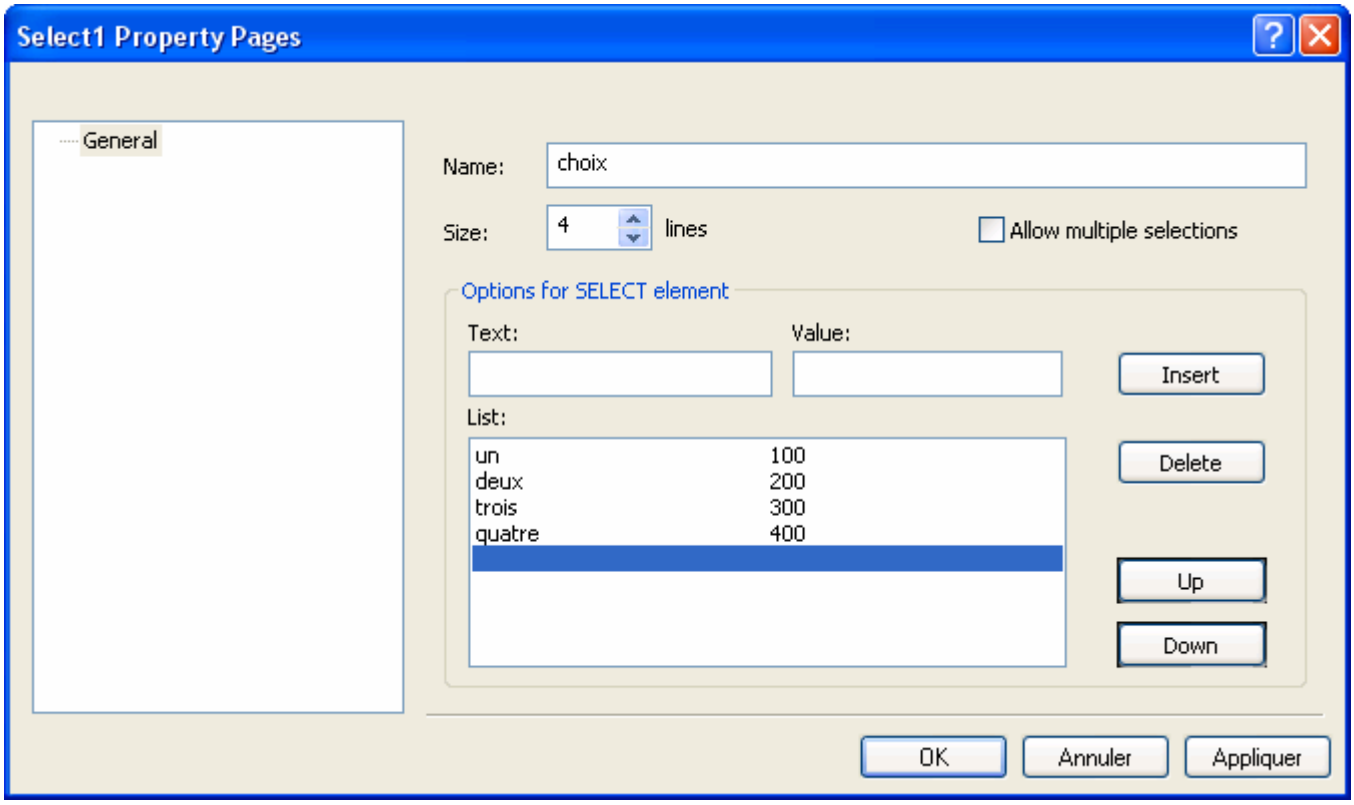
Second exemple d'utilisation de HtmlControls : afficher une liste de sélection dans une page web.

On choisi le HtmlControls **Select** dans la Toolbox et on le dépose sur la page :	On change sa propriété size à 4 (ce qui permet de définir une liste de sélection possédant 4 éléments) :





En cliquant avec le bouton droit de souris sur l'image déposée et en choisissant le champ propriétés dans le menu popup, on obtient une fenêtre permettant de modifier la liste :

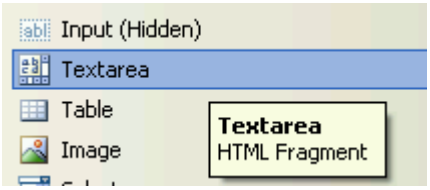
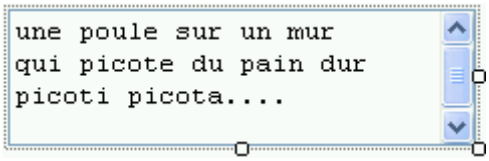


Entité visuelle :

Le code source HTML obtenu est insérable dans une page :

```
<html>
<head>
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1">
    <div>
        <select id="Select1" name="choix" style="height: 84px" size="4">
            <option selected="selected" value="100">un</option>
            <option value="200">deux</option>
            <option value="300">trois</option>
            <option value="400">quatre</option>
        </select>
    </div>
    </form>
</body>
</html>
```

Dernier exemple d'utilisation de HtmlControls : utiliser un TextArea dans une page web.

Onglet HTML de la Toolbox :	**Entité visuelle** :
	

Le code source HTML obtenu est insérable dans une page :

```
<html>
<head>
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1">
        <textarea id="TextArea1" rows="4" style="width: 227px">
une poule sur un mur
qui picote du pain dur
picoti picota....
        </textarea>
    </form>
</body>
</html>
```

5°) Utiliser Visual Studio pour générer des appels au javascript

L'environnement de programmation Visual Studio permet aussi de créer visuellement les réactions événementielles d'une entité de votre page HTML avec des fonctions javascript.

On construit une page HTML :

Contenant par exemple un <INPUT> du type "bouton" :

```
<input id="Button1" type="button" value="button" />
```

button

Contenant aussi un <INPUT> du type "text" :

```
<input id="Text1" type="text" />
```

On peut rendre la page HTML sensible au click de souris chez le client :

En double cliquant sur l'image button de l'INPUT bouton, Visual Studio génère deux éléments :

1- un squelette de fonction javascript
2- le tag associé à l'événement onclick dans la balise INPUT

```
<html>
<head runat="server">
    <title>Untitled Page</title>
<script language="javascript"
type="text/javascript">
<!--

function Button1_onclick() {

}

// -->
</script>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <input id="Button1" type="button" value="button"
            language="javascript" onclick="return
Button1_onclick()" />
        <br />
        <br />
        <input id="Text1" type="text" />
        </div>
    </form>
</body>
</html>
```

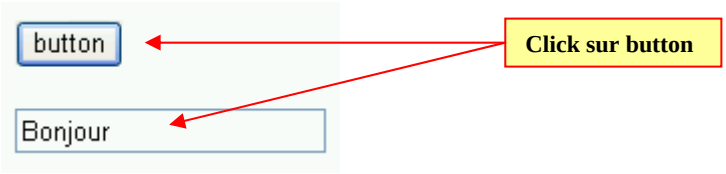
Squelette javascript

tag onclick

Action événementielle proposée : en double cliquant sur l'entité `Button1"`, la fonction Button1_Click() renvoie le texte "Bonjour" dans l'entité `"Text1"`.

```
function Button1_onclick() {
 document.form1.Text1.value = "Bonjour";
}
```

code javascript



6°) Utilisez du javascript pour onmouseover et onmousemove

Afin de ne pas surcharger inutilement le serveur, on programme les événements de souris (over et move en particulier) côté client avec du javascript et avec des scripts C# côté serveur :

Le code source HTML obtenu est insérable dans une page :

```
<html>
<head>
    <title>Untitled Page</title>
```

```
    <script language="javascript">
    <!--
       function Button1_onmouseover() {
          window.status ="Button1_onmouseover : X ="+
                                    window.event.x+" , Y = "+window.event.y;
        }
        function Button1_onmouseleave() {
          window.status ="Clear";
        }
        function Text1_onmousemove() {
          window.status ="Text1_onmousemove : X ="+
                                     window.event.x+" , Y = "+window.event.y;
        }
    // -->
    </script>
```

```
</head>
<body>
    <form id="form1" >
        <div>
```

```
            <input id="Button1" type="button" value="button"
                       onmouseleave="Button1_onmouseleave()"
                       onmouseover="Button1_onmouseover()"  />
```

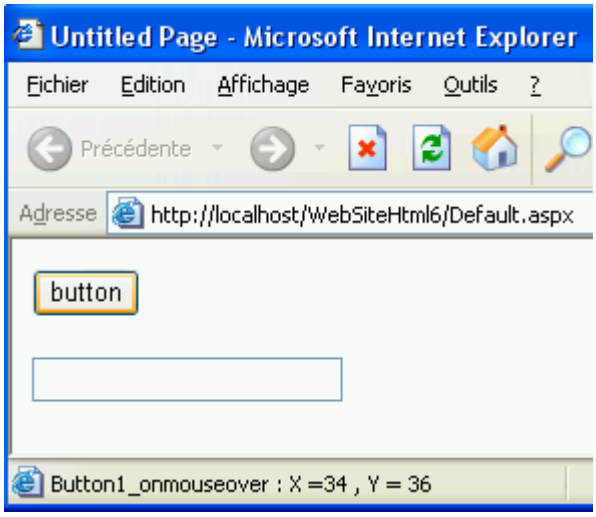
```
            <br />
            <br />
```

```
            <input id="Text1" onmousemove="Text1_onmousemove()" type="text"
/>
```

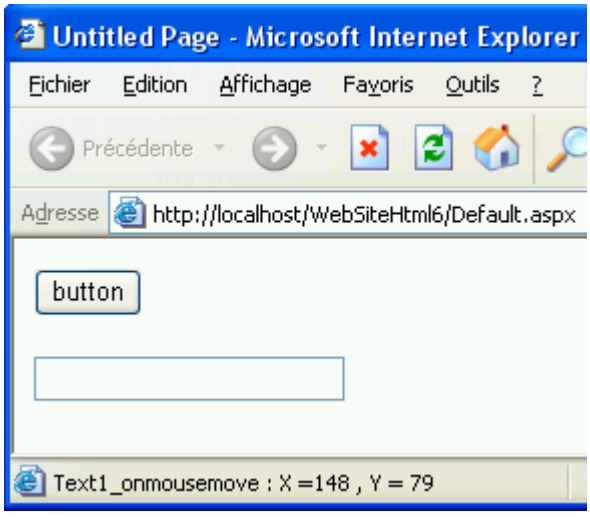
```
        </div>
    </form>
</body>
</html>
```

Les actions visualisées côtés client

```
onmouseover="Button1_onmouseover()"  :
```



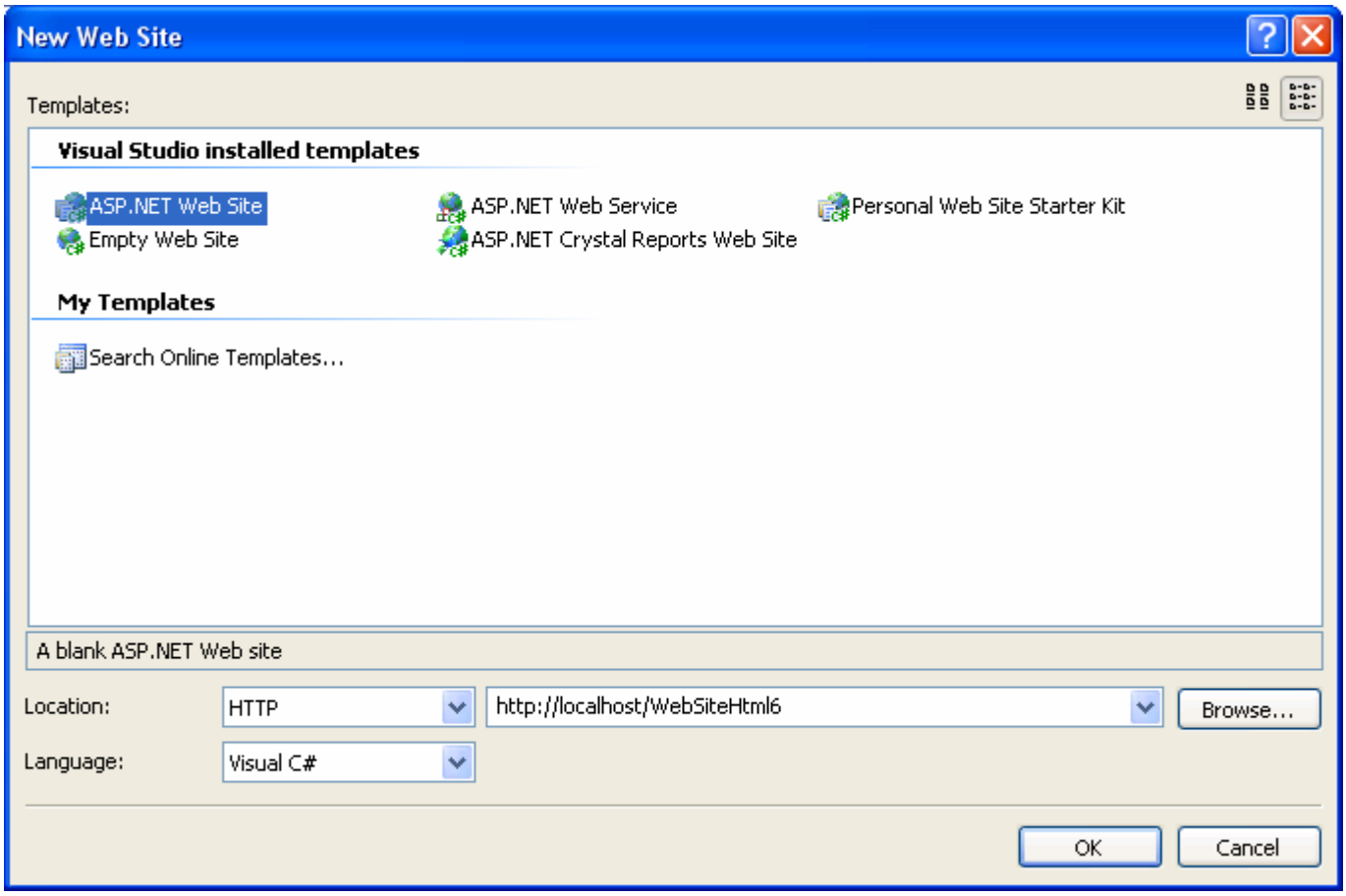

```
onmousemove="Text1_onmousemove()" :
```



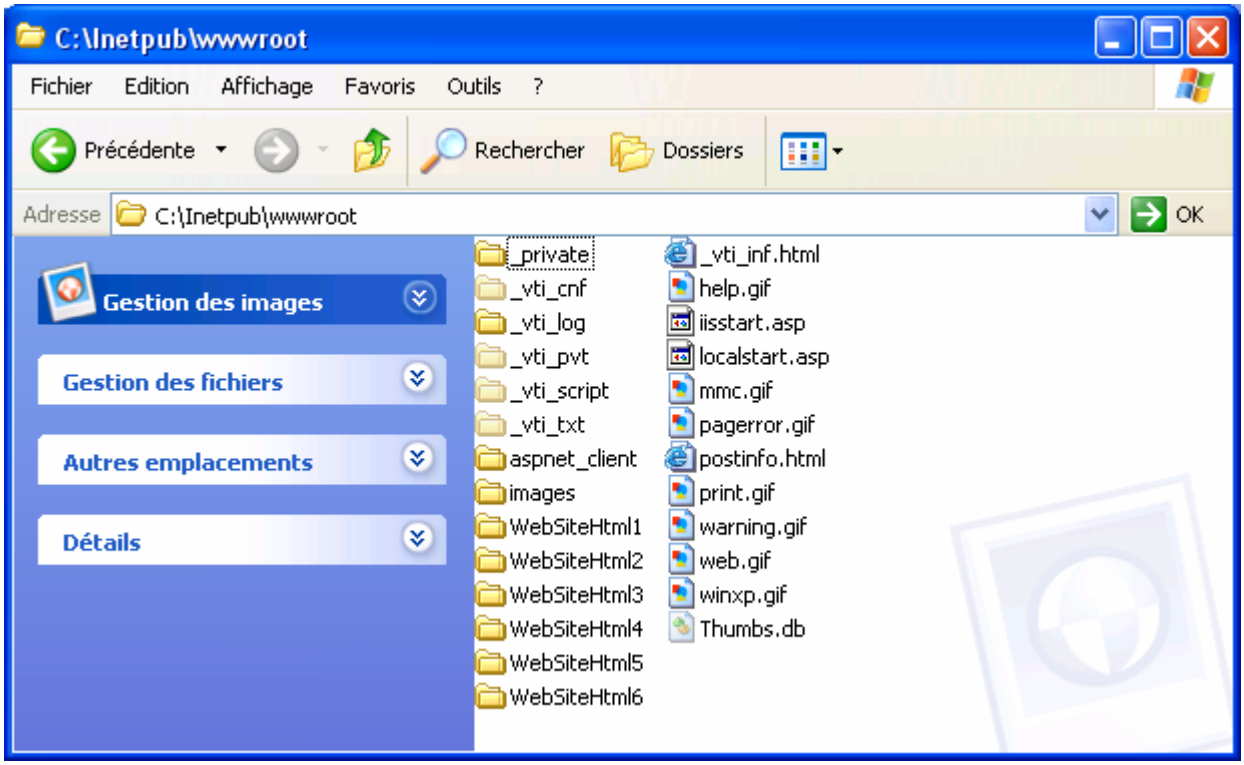
Conclusion :

Une page ASP .net 2.0 fonctionne et réagit comme une page classique HTML.

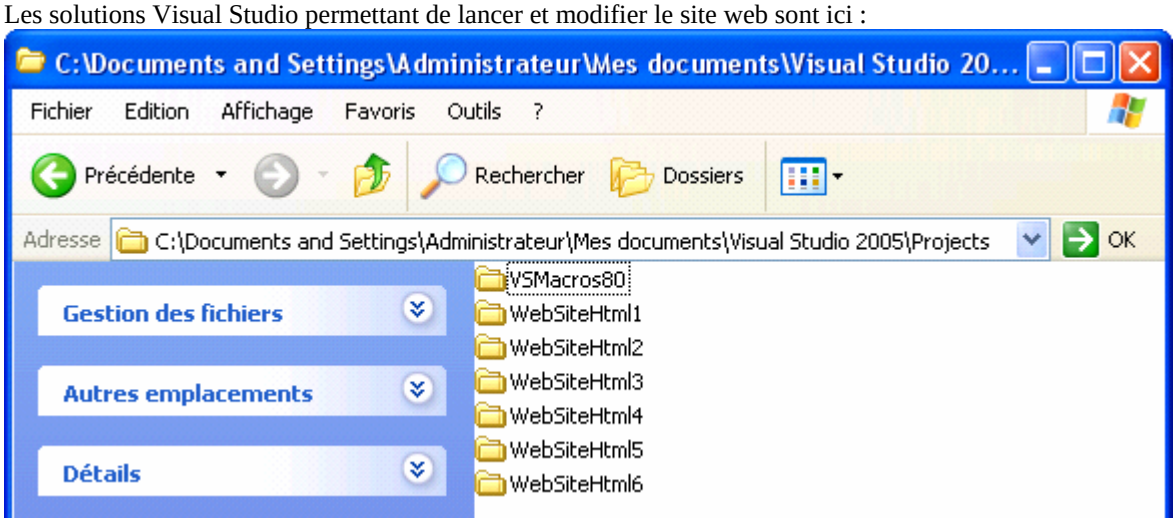
Où sont localisés les fichiers des sites web construits



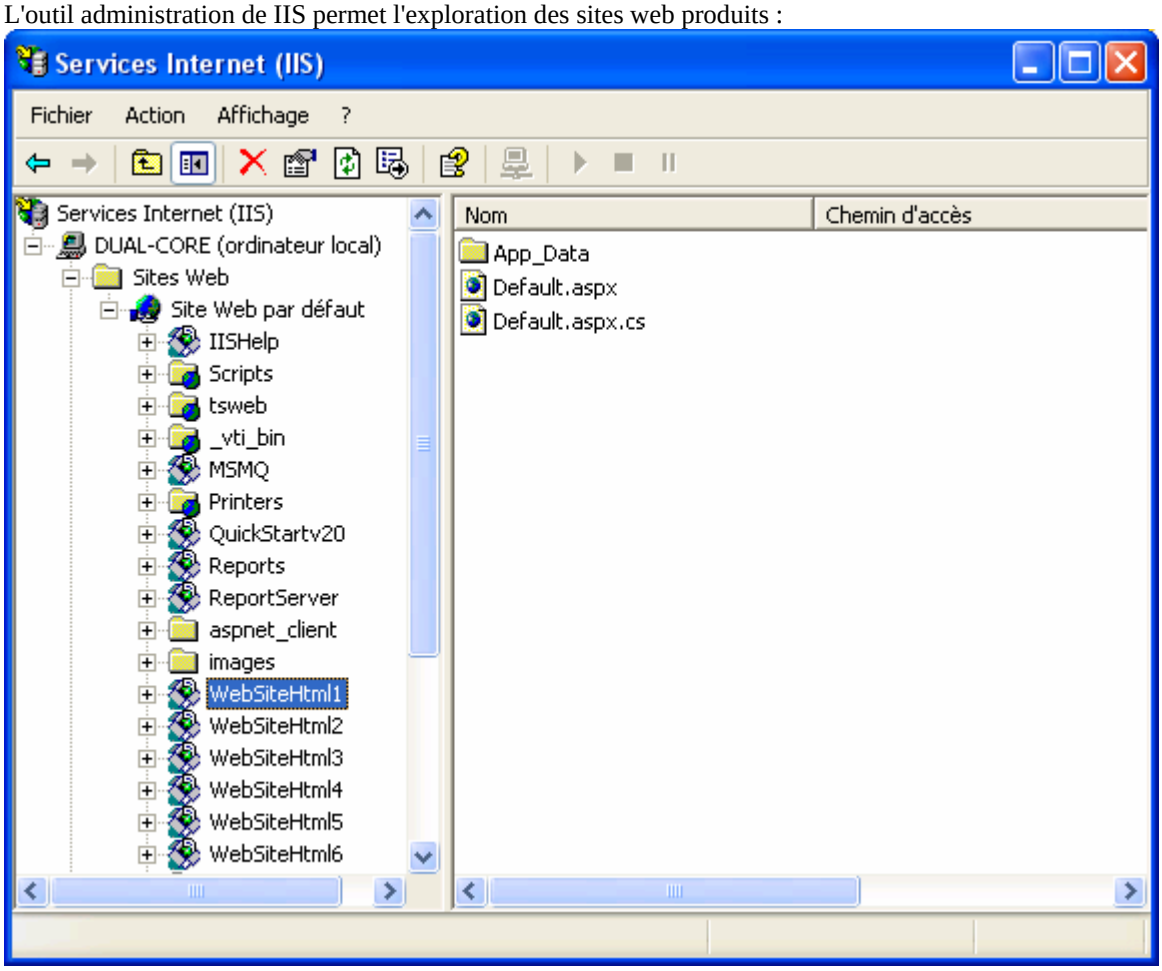
*On a choisi la localisation http (sur le localhost, rangement dans **C:\Inetpub\wwwroot**)*



Les solutions Visual Studio permettant de lancer et modifier le site web sont ici :



L'outil administration de IIS permet l'exploration des sites web produits :



Page web écrite avec des contrôles serveurs

Plan général:

Page web écrite uniquement avec des contrôles serveurs

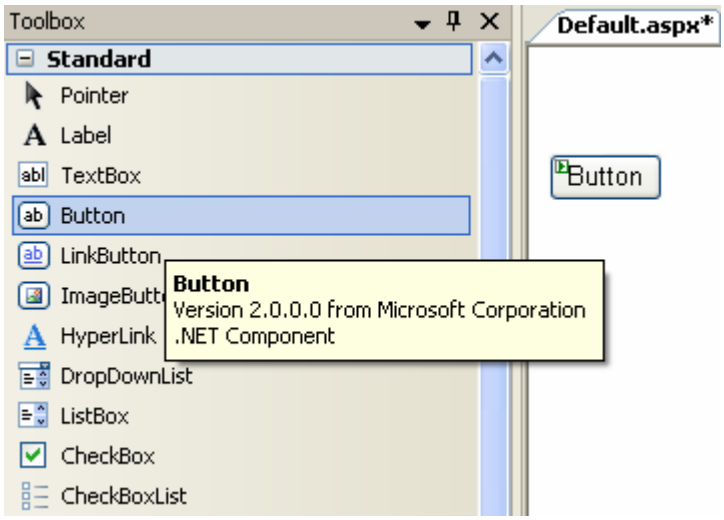
1°) Construire une page Asp.net côté serveur avec Visual Studio
2°) Une page Asp.net côté serveur peut engendrer du javascript
3°) Les événements gérés côté serveur sont dans une page séparée (code-behind)
4°) Les événements gérés côté serveur sont dans un script inclus (code-inline)
5°) Les événements côté serveur permettent des communications entre les contrôles

Les contrôles serveurs : ce sont des objets qui se programment et s'exécutent côté serveur. Ils servent à représenter des éléments d'IHM dans une page web (éditeurs mono et multi lignes, images, boutons, …). Pour déclarer et utiliser de tels contrôles, il suffit d'ajouter dans le code source d'une page Asp.Net (une page Asp.Net = un fichier **XXX.aspx**) les balises HTML-like spécifiques à ces objets de contrôles serveurs.

- Lors de l'exécution sur le serveur, ces contrôles serveurs génèrent automatiquement un contenu HTML.
- Le code source JHTML décrivant un contrôle serveur contient, dans la balise du contrôle serveur, l'attribut **runat** positionné sur la valeur **server** (attribut **runat = server**). Cet attribut **runat** spécifie que le code inclus dans le bloc de script s'exécute sur le serveur et non chez le client (par défaut l'exécution a lieu chez le client).
- Ces contrôles serveurs sont des objets instanciés à partir de classes situés dans le **namespace** suivant : **System.Web.UI.WebControls**.
- Il est aussi possible de transformer en contrôles serveurs les contrôles HTML côté client (contrôles du namespace **System.Web.UI.HtmlControls**) d'une page Asp.Net déjà écrite.
- Un contrôle serveur est généralement décrit dans une page Asp.net grâce à une balise spécifique **<asp: …>**.

1°) Construire une page Asp.net côté serveur avec Visual Studio :

Visual Studio permet de déposer visuellement des contrôles serveurs afin de construire son IHM, l'IDE engendre alors le code Asp.Net associé. Par exemple un bouton peut être créé à partir de l'onglet standard de la Toolbox de Visual Studio dans une page nommée **Default.aspx** :



Code partiel engendré par Visual Studio pour ce bouton : Button

```
<form id="form1" runat="server">
        <asp:Button ID="Button1" runat="server"
Text="Button" />
</form>
```

La Toolbox contient plusieurs autres onglets facilitant la création visuelle de pages Asp.Net avec de nombreux outils :

	Seul le code HTML est envoyé au navigateur du client, c'est

donc Visual Studio qui transcrit les balises et le code Asp.Net en du code classique reconnaissable par n'importe quel navigateur Internet.

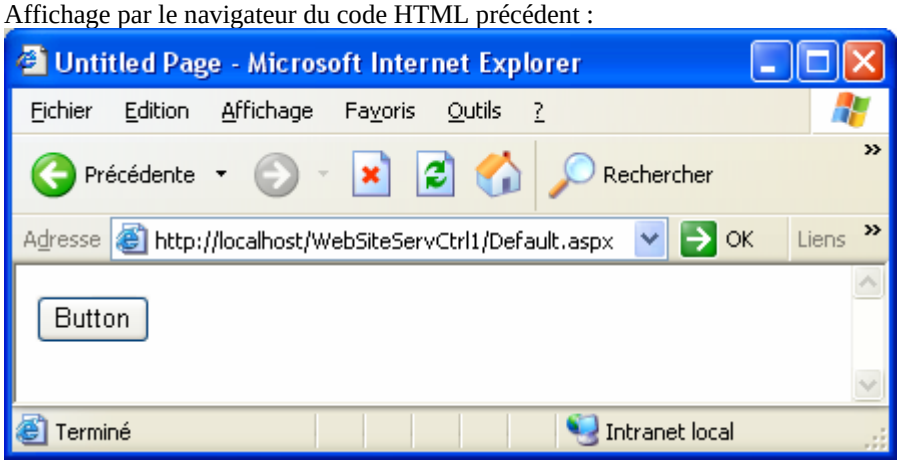
Code Asp.Net du bouton précédent :

```
<form id="form1" runat="server">
        <asp:Button ID="Button1" runat="server"
Text="Button" />
</form>
```

Code HTML engendré pour le bouton précédent :

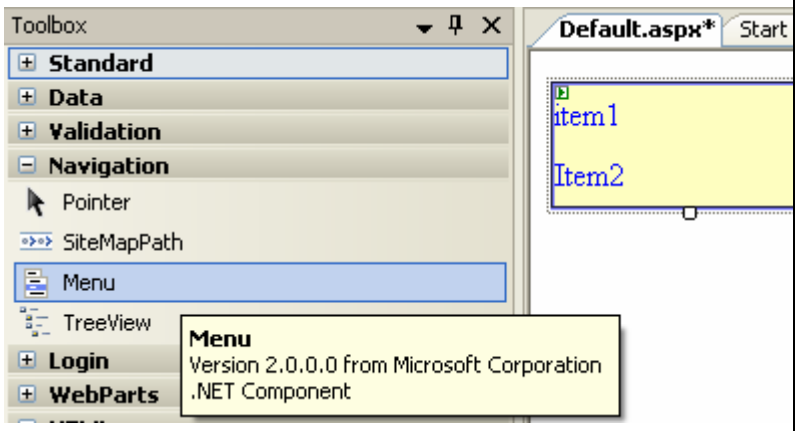
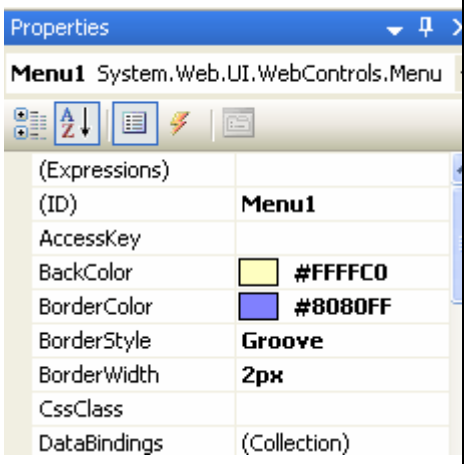
```
<body>
    <form name="form1" method="post" action="Default.aspx"
id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTQ2OTkzNDMyMWRkNOC5wTqqk2noztuYBp2ZpDkCmmE=" />
</div>
    <div>
        <input type="submit" name="Button1" value="Button"
id="Button1" />
    </div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAgLp8MT+DQKM54rGBvbawE34cFV6yTqH8kCIHOAvnsVn" />
</div></form>
</body>
```

Affichage par le navigateur du code HTML précédent :

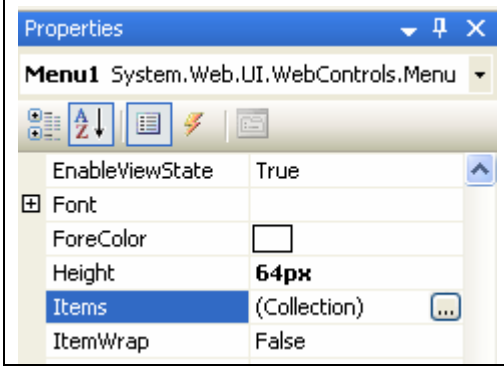


2°) Une page Asp.net côté serveur peut engendrer du javascript :

Visual Studio engendre lui-même du code javascript à partir de certains objets de classes **System.Web.UI.WebControls** comme par exemple la classe **Menu** qui permet d'afficher un menu dans une page web :

La Toolbox et le menu web déposé	L'inspecteur de propriété du menu
	

Il est possible de créer visuellement une arborescence de sous-menus dans ce menu grâce à la propriété **Items** de la classe Menu :



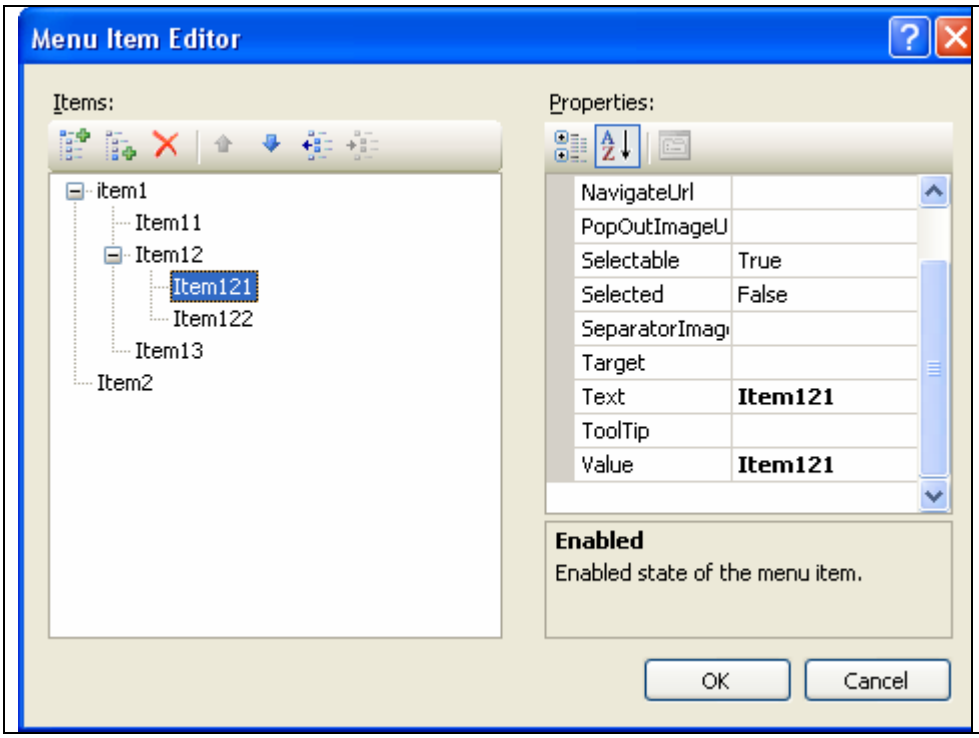
public MenuItemCollection Items { get; }

Cette propriété représente une collection d'objets de classe MenuItem.

public sealed class MenuItemCollection : ICollection, IEnumerable, IStateManager
Cette classe contient un indexeur permettant d'accéder à chaque objet MenuItem :
`public MenuItem this [ int index ] { get; }`

public sealed class MenuItem : IStateManager, ICloneable
Cette classe possède une propriété Text contenant le libellé du sous-menu.
public string Text { get; set; }

Il est possible d'utiliser un éditeur de propriété de Visual Studio afin de construire visuellement l'arborescence de sous-menu :



Le menu sur la page web :

Code Asp.Net du Menu ainsi construit :

```
    <form id="form1" runat="server">
    <div>
         <asp:Menu ID="Menu1" runat="server" BackColor="#FFFFC0"
BorderColor="#8080FF"
                  BorderStyle="Groove" BorderWidth="2px" Width="140px"
Height="64px">
            <Items>
                <asp:MenuItem Text="item1" Value="item1">
                    <asp:MenuItem Text="Item11"
Value="Item11"></asp:MenuItem>
                    <asp:MenuItem Text="Item12" Value="Item12">
                        <asp:MenuItem Text="Item121"
Value="Item121"></asp:MenuItem>
                        <asp:MenuItem Text="Item122"
Value="Item122"></asp:MenuItem>
                    </asp:MenuItem>
                    <asp:MenuItem Text="Item13"
Value="Item13"></asp:MenuItem>
                </asp:MenuItem>
                <asp:MenuItem Text="Item2" Value="Item2"></asp:MenuItem>
            </Items>
           <DynamicMenuItemStyle BackColor="#FFFFC0" BorderColor="#8080FF"
                                 BorderStyle="Ridge"
BorderWidth="2px" />
        </asp:Menu>
    </div>
    </form>
```

Code HTML engendré pour le Menu précédent : (*remarquez les appels au javascript généré*)

```
<head><title>Untitled Page</title>
```

```
<style type="text/css">
      .Menu1_0 { background-color: white; visibility:hidden; display:none;
position:absolute; left:0px;top:0px; }
      .Menu1_1 { text-decoration:none; }
      .Menu1_2 { background-color:#FFFFC0;border-color:#8080FF;border-
width:2px;border-style:Groove;height:64px;width:140px; }
      .Menu1_3 {  }
      .Menu1_4 { background-color:#FFFFC0;border-color:#8080FF;border-
width:2px;border-style:Ridge; }

</style></head>

<body>
<form name="form1" method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUJNTc3OTkxNTEwZGSArJ/9WCLlkb/ok842mKSeYFGUyg==" />
</div>
```

```
<script type="text/javascript">
<!--
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() !=
false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value =
eventArgument;
        theForm.submit();
    }
}
// -->
</script>
```

```
<script
src="/WebSiteServCtrl2/WebResource.axd?d=BQ2832wAAdlkKigxtyKV4Q2&amp;t=632738
879195937500" type="text/javascript">
</script>
```

Fichier de code javascript de gestion d'une fiche (déjà écrit cf. annexe).

```
<script
src="/WebSiteServCtrl2/WebResource.axd?d=stc__7z_BWk1ov4p1EicdA2&amp;t=632738
879195937500" type="text/javascript">
</script>
```

Fichier de code javascript de gestion de menus (déjà écrit cf. annexe).

Ces fichiers de code javascript nommés WebResource.axd?d=….. sont auto générés par Visuel Studio et se trouvent dans "C:\Documents and Settings\Administrateur\Local Settings\Temporary Internet Files" (ici le nom d'utilisateur est "Administrateur") :

C:\Documents and Settings\votreNomUtilisateur\Local Settings\Temporary Internet Files:

Nom	Adresse Internet
Default.aspx	view-source:http://localhost/WebSiteServCtrl2/Default
WebResource.axd?d=40dWBmhSJB94j4P_hX-jZNopyRqFAQF5gdcZcekCEfk...	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=40dWBmhSJB94j4P_hX-jZMgnL1NYlJ1X5KRooaFMVYk...	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=t0BwhNHjAD7Z4AcKiXLHSWdrpiqgp9jN0a6_Au0mQVE...	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=JBDdLBfoNmBArrcf-A6P2g2&t=632738879195937500	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=B8o8yLHFIkai-UmHjueg2_fiFIVM8HAr3u-akpSun5U1&...	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=LN1hR4BWAG0U64YM2MVYbNfLaEVdlrAGsib2pts53rk...	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=stc__7z_BWk1ov4p1EicdA28&t=632738879195937500	http://localhost/WebSiteServCtrl2/WebResource.axd?
WebResource.axd?d=u9eZty7kcs7SiKaCaDTrlg28&t=632738879195937500	http://localhost/WebSiteServCtrl2/WebResource.axd?
Default.aspx	http://localhost/WebSiteServCtrl2/Default.aspx

```
<div>
<a href="#Menu1_SkipLink">
<img alt="Skip Navigation Links"
src="/WebSiteServCtrl2/WebResource.axd?d=JBDdLBfoNmBArrcf-
A6P2g2&amp;t=632738879195937500" width="0" height="0" style="border-
width:0px;"/>
</a>
<table id="Menu1" class="Menu1_2" cellpadding="0" cellspacing="0" border="0">
<tr onmouseover="Menu_HoverStatic(this)" onmouseout="Menu_Unhover(this)"
onkeyup="Menu_Key(this)" id="Menu1n0">
            <td><table cellpadding="0" cellspacing="0" border="0"
width="100%">
                  <tr>
                        <td style="white-space:nowrap;width:100%;"><a
class="Menu1_1"
href="javascript:__doPostBack('Menu1','item1')">item1</a></td><td
style="width:0;"><img
src="/WebSiteServCtrl2/WebResource.axd?d=t0BwhNHjAD7Z4AcKiXLHSWdrpiqgp9jN0a6_
Au0mQVE1&amp;t=632738879195937500" alt="Expand item1" style="border-
style:none;vertical-align:middle;" /></td>
                  </tr>
            </table></td>
      </tr><tr onmouseover="Menu_HoverStatic(this)"
onmouseout="Menu_Unhover(this)" onkeyup="Menu_Key(this)" id="Menu1n1">
            <td><table cellpadding="0" cellspacing="0" border="0"
width="100%">
                  <tr>
                        <td style="white-space:nowrap;width:100%;"><a
class="Menu1_1"
href="javascript:__doPostBack('Menu1','Item2')">Item2</a></td>
                  </tr>
            </table></td>
      </tr>
</table><div id="Menu1n0Items" class="Menu1_0">
<table border="0" cellpadding="0" cellspacing="0">
<tr onmouseover="Menu_HoverDynamic(this)" onmouseout="Menu_Unhover(this)"
onkeyup="Menu_Key(this)" id="Menu1n2">
<td><table class="Menu1_4" cellpadding="0" cellspacing="0" border="0"
width="100%">
<tr>
<td style="white-space:nowrap;width:100%;"><a class="Menu1_1 Menu1_3"
href="javascript:__doPostBack('Menu1','item1\\Item11')">Item11</a></td>
</tr>
</table></td>
</tr><tr onmouseover="Menu_HoverDynamic(this)"
onmouseout="Menu_Unhover(this)" onkeyup="Menu_Key(this)" id="Menu1n3">
```

```
<td><table class="Menu1_4" cellpadding="0" cellspacing="0" border="0"
width="100%">
<tr>
<td style="white-space:nowrap;width:100%;"><a class="Menu1_1 Menu1_3"
href="javascript:__doPostBack('Menu1','item1\\Item12')">Item12</a></td><td
style="width:0;"><img
src="/WebSiteServCtrl2/WebResource.axd?d=t0BwhNHjAD7Z4AcKiXLHSWdrpiqgp9jN0a6_
Au0mQVE1&amp;t=632738879195937500" alt="Expand Item12" style="border-
style:none;vertical-align:middle;" /></td>
</tr>
</table></td>
</tr><tr onmouseover="Menu_HoverDynamic(this)"
onmouseout="Menu_Unhover(this)" onkeyup="Menu_Key(this)" id="Menu1n4">
<td><table class="Menu1_4" cellpadding="0" cellspacing="0" border="0"
width="100%">
<tr>
<td style="white-space:nowrap;width:100%;"><a class="Menu1_1 Menu1_3"
href="javascript:__doPostBack('Menu1','item1\\Item13')">Item13</a></td>
</tr>
</table></td>
</tr>
</table><div class="Menu1_4 Menu1_0" id="Menu1n0ItemsUp"
onmouseover="PopOut_Up(this)" onmouseout="PopOut_Stop(this)" style="text-
align:center;">
<img
src="/WebSiteServCtrl2/WebResource.axd?d=LN1hR4BWAG0U64YM2MVYbNfLaEVdlrAGsib2
pts53rk1&amp;t=632738879195937500" alt="Scroll up" />
</div><div class="Menu1_4 Menu1_0" id="Menu1n0ItemsDn"
onmouseover="PopOut_Down(this)" onmouseout="PopOut_Stop(this)" style="text-
align:center;">
            <img src="/WebSiteServCtrl2/WebResource.axd?d=B8o8yLHFIkai-
UmHjueg2_fiFIVM8HAr3u-akpSun5U1&amp;t=632738879195937500" alt="Scroll down"
/>
      </div>
</div><div id="Menu1n3Items" class="Menu1_0">
<table border="0" cellpadding="0" cellspacing="0">
<tr onmouseover="Menu_HoverDynamic(this)" onmouseout="Menu_Unhover(this)"
onkeyup="Menu_Key(this)" id="Menu1n5">
<td><table class="Menu1_4" cellpadding="0" cellspacing="0" border="0"
width="100%">
<tr>
<td style="white-space:nowrap;width:100%;"><a class="Menu1_1 Menu1_3"
href="javascript:__doPostBack('Menu1','item1\\Item12\\Item121')">Item121</a><
/td>
</tr>
</table></td>
</tr><tr onmouseover="Menu_HoverDynamic(this)"
onmouseout="Menu_Unhover(this)" onkeyup="Menu_Key(this)" id="Menu1n6">
<td><table class="Menu1_4" cellpadding="0" cellspacing="0" border="0"
width="100%">
                        <tr>
<td style="white-space:nowrap;width:100%;"><a class="Menu1_1 Menu1_3"
href="javascript:__doPostBack('Menu1','item1\\Item12\\Item122')">Item122</a><
/td>
</tr>
</table></td>
</tr>
</table><div class="Menu1_4 Menu1_0" id="Menu1n3ItemsUp"
onmouseover="PopOut_Up(this)" onmouseout="PopOut_Stop(this)" style="text-
align:center;">
```

```
<img
src="/WebSiteServCtrl2/WebResource.axd?d=LN1hR4BWAG0U64YM2MVYbNfLaEVdlrAGsib2
pts53rk1&amp;t=632738879195937500" alt="Scroll up" />
</div><div class="Menu1_4 Menu1_0" id="Menu1n3ItemsDn"
onmouseover="PopOut_Down(this)" onmouseout="PopOut_Stop(this)" style="text-
align:center;">
<img src="/WebSiteServCtrl2/WebResource.axd?d=B8o8yLHFIkai-
UmHjueg2_fiFIVM8HAr3u-akpSun5U1&amp;t=632738879195937500" alt="Scroll down"
/>
</div>
</div><a id="Menu1_SkipLink"></a>
</div>
<div>
      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWCAKdo674AQLh6ozdAwL+4ozdAwKPrOzfBwKPrMC0DwKPrNTpBAKjqPjVDgKjqOTwCa
hFukoD2hz8YEfJjmjjDVyK2FwP" />
</div>
```

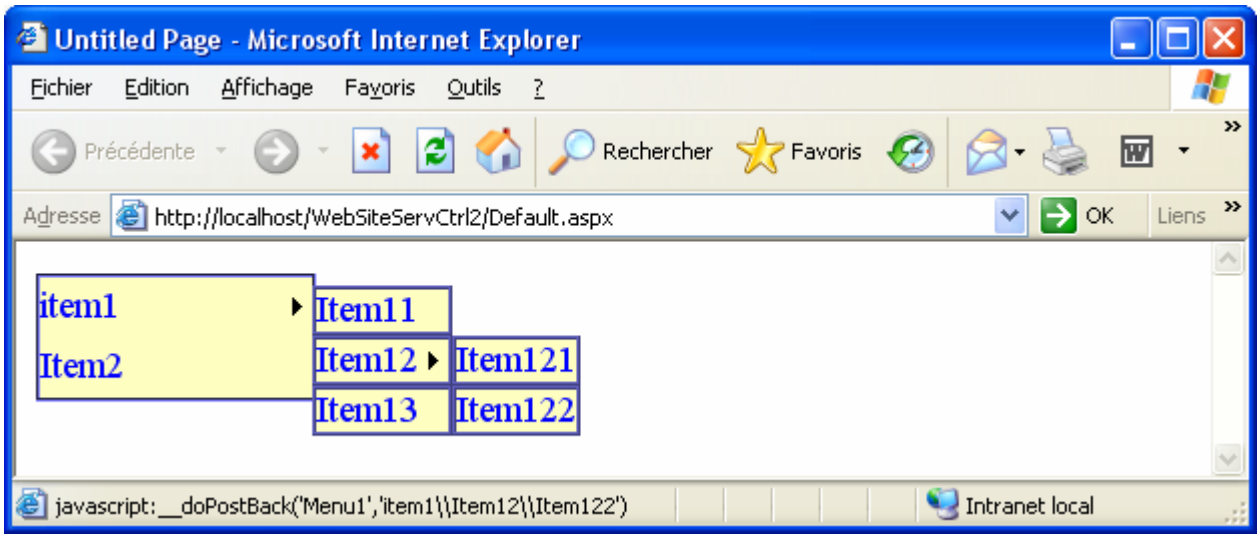
```
<script
type="text/javascript">
<!--
var Menu1_Data = new
Object();
Menu1_Data.disappearAfter
= 500;
Menu1_Data.horizontalOffs
et = 0;
Menu1_Data.verticalOffset
= 0;
// -->
</script>
</form>
</body>
```

- Les fichiers contenant des fonctions javascript servant à gérer la fiche et les menus "`WebResource.axd?d=BQ2832wAAdlkKigxtyKV4Q2&t=632738879195937500`" et "`WebResource.axd?d=stc__7z_BWk1ov4p1EicdA2&t=632738879195937500`" peuvent être trouvés dans le dossier des fichiers temporaires Internet :

  C:\Documents and Settings\VotreNom\Local Settings\Temporary Internet Files

- Si l'on sauvegarde la page HTML à partir du navigateur, on récupère alors localement, une copie de ces deux fichiers (ici `WebResource(1).axd` et `WebResource.axd`) :

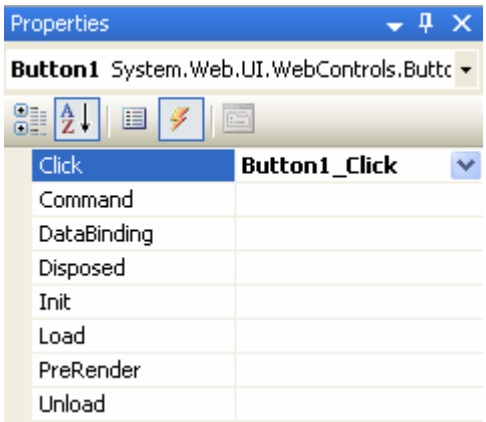
Affichage par le navigateur du code HTML précédent :



3°) Les événements gérés côté serveur sont dans une page séparée (code-behind) :

Soit un contrôle de classe **System.Web.UI.WebControls.Button** :

Ce contrôle est sensible à 8 événements, il suffit de programmer sa réaction à un des événements comme par exemple le click de l'utilisateur sur ce bouton :

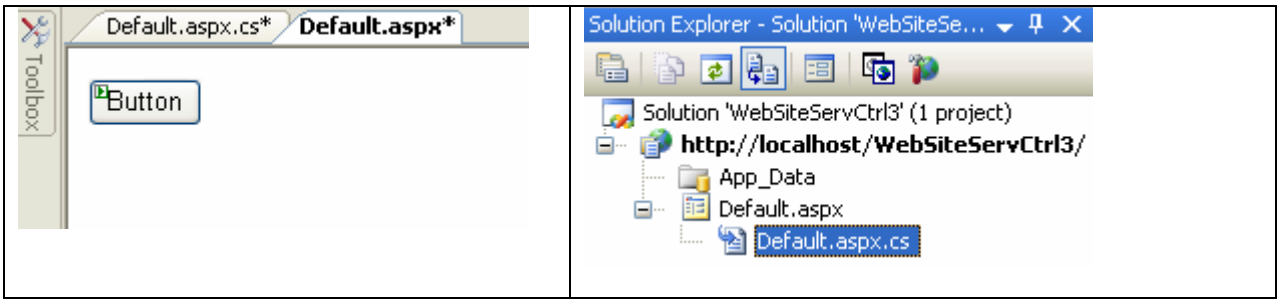


Si la forme visuelle du bouton se trouve déposée dans une page nommée **Default.aspx**, le code C# du gestionnaire de réaction à chacun des 8 événements du bouton se trouve dans une autre page associée nommée **Default.aspx.cs**.

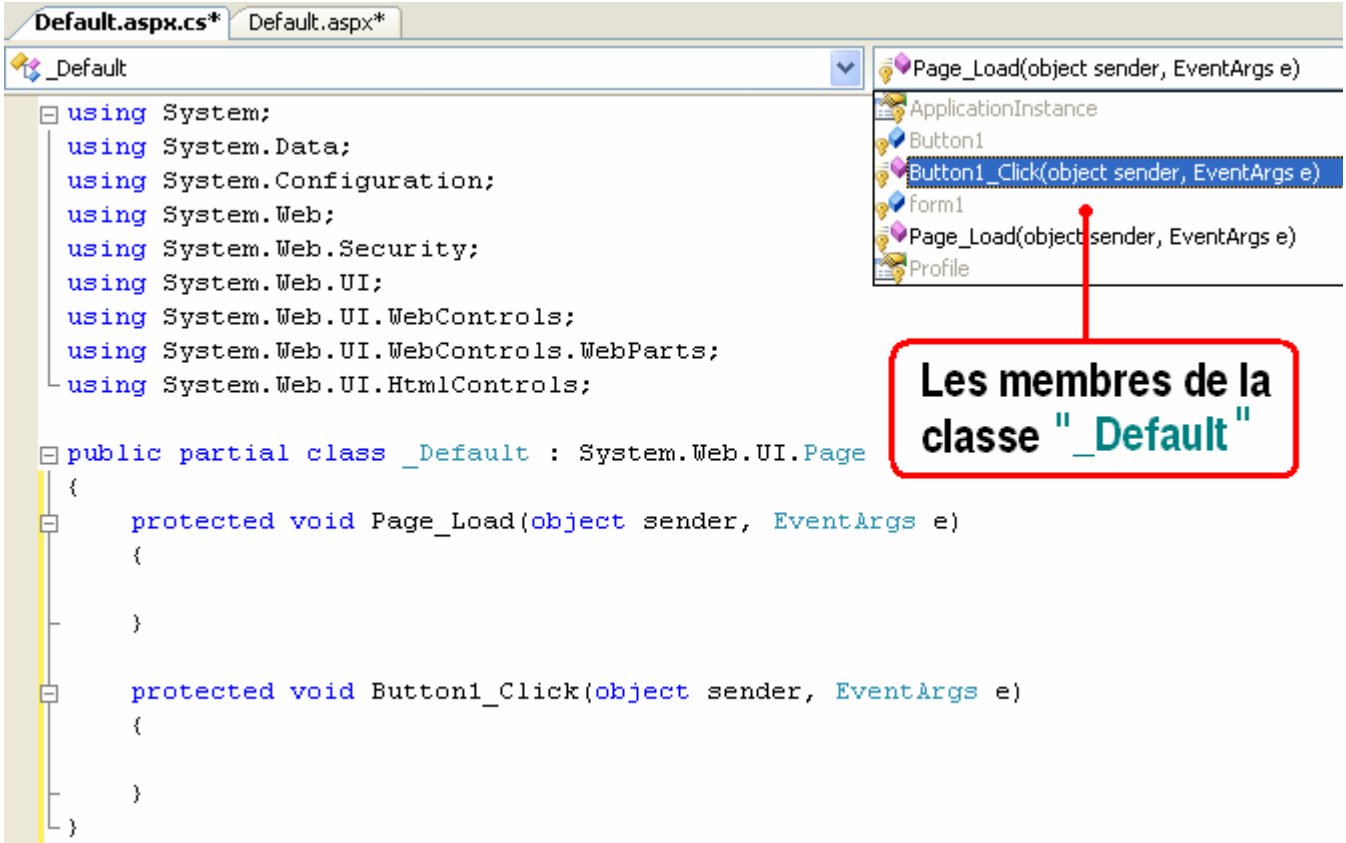
Remarque :

> Cette page de code reprend le même nom que la page à laquelle elle est associée et rajoute le suffixe **cs** pour du code de réaction du type **C#**, le suffixe est **vb** (**Default.aspx.vbs**) pour du code de type **VB.Net**, **js** pour le java **J#** et ainsi de suite pour tout langage supporté par la plateforme .Net.

La page visuelle du bouton	La page de code associée dans l'explorateur



Lors de la demande de création du gestionnaire de réaction au click de souris sur le bouton, Visual Studio engendre une classe partielle nommée _Default qui hérite de la classe **System.Web.UI.Page** : cette technique est appelée **code-behind** (code en arrière, code en coulisse, code caché…), en outre la classe partielle permet d'écrire plusieurs fichiers de code et de les assembler en une seule classe à la compilation.



La classe **Page** est associée à des fichiers de suffixe **aspx**. Ces fichiers sont compilés lors de l'exécution comme des objets de type **Page** et stockés dans la mémoire cache du serveur.

Tous les RAD qui supportent la construction de pages web avec la technique du **code-behind** dans .Net doivent créer une classe héritant de la classe **Page**.

Dans la classe de notre exemple, nous trouvons deux squelettes de gestionnaires d'événements :

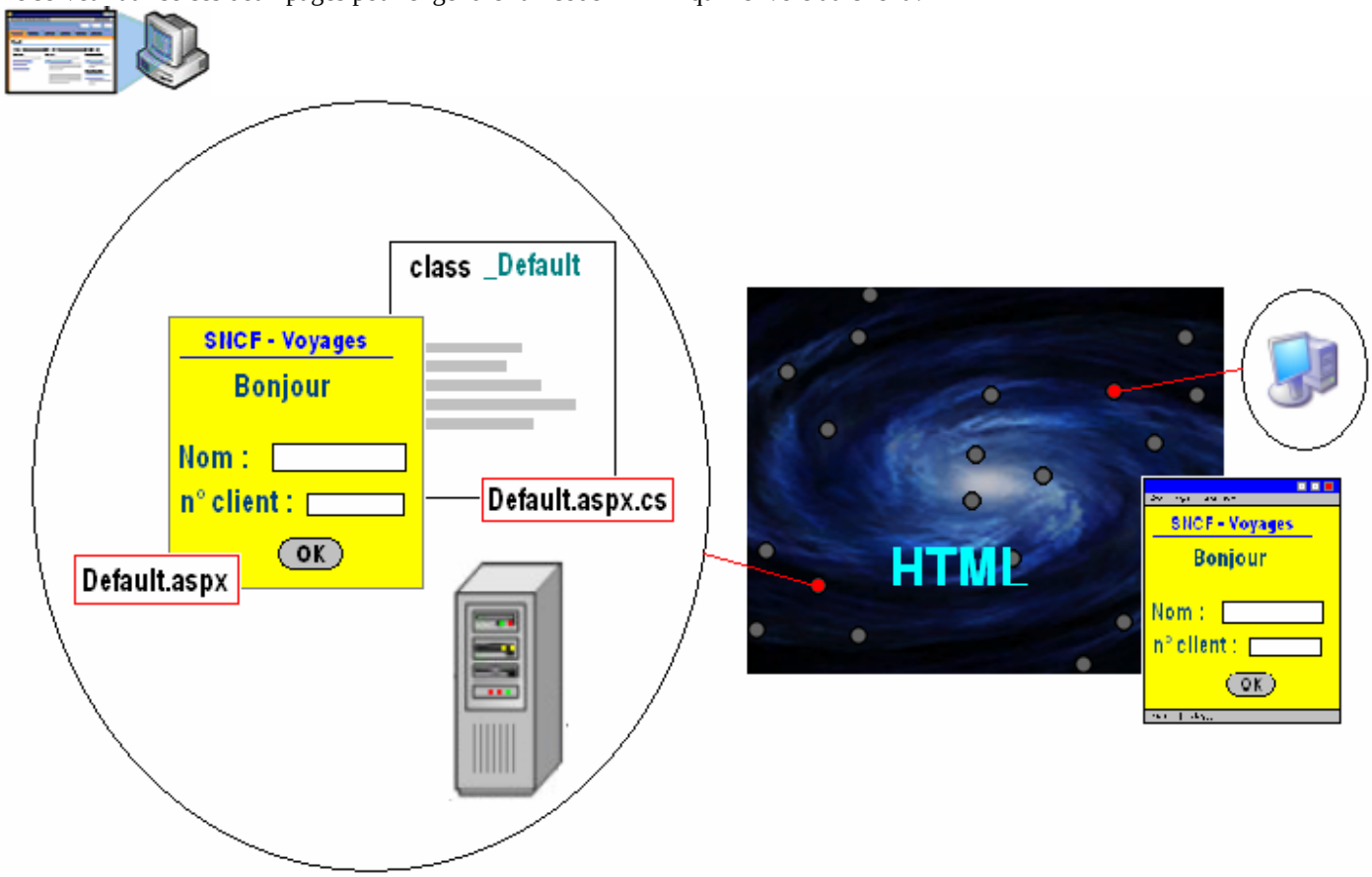
- Le premier "protected void Page_Load(object sender, EventArgs e)" est un gestionnaire de l'événement "Load" de la page elle-même,
- le second "protected void Button1_Click(object sender, EventArgs e)" est le gestionnaire de l'événement "click" du bouton sur la page.

L'événement "Load" a lieu à chaque chargement de la page à partir du serveur c'est l'un des 15 événements possibles pouvant avoir lieu sur une page web :

Public Events

	Name		Name
	AbortTransaction		Load
	CommitTransaction		LoadComplete
	DataBinding		PreInit
	Disposed		PreLoad
	Error		PreRender
	Init		PreRenderComplete
	InitComplete		SaveStateComplete
			Unload

Le serveur utilise ces deux pages pour engendrer un code HTML qu'il envoie au client :



Code Asp.Net de la page Default.aspx ainsi construite :

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
      <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
Text="Button" />
    </div>
    </form>
</body>
</html>
```

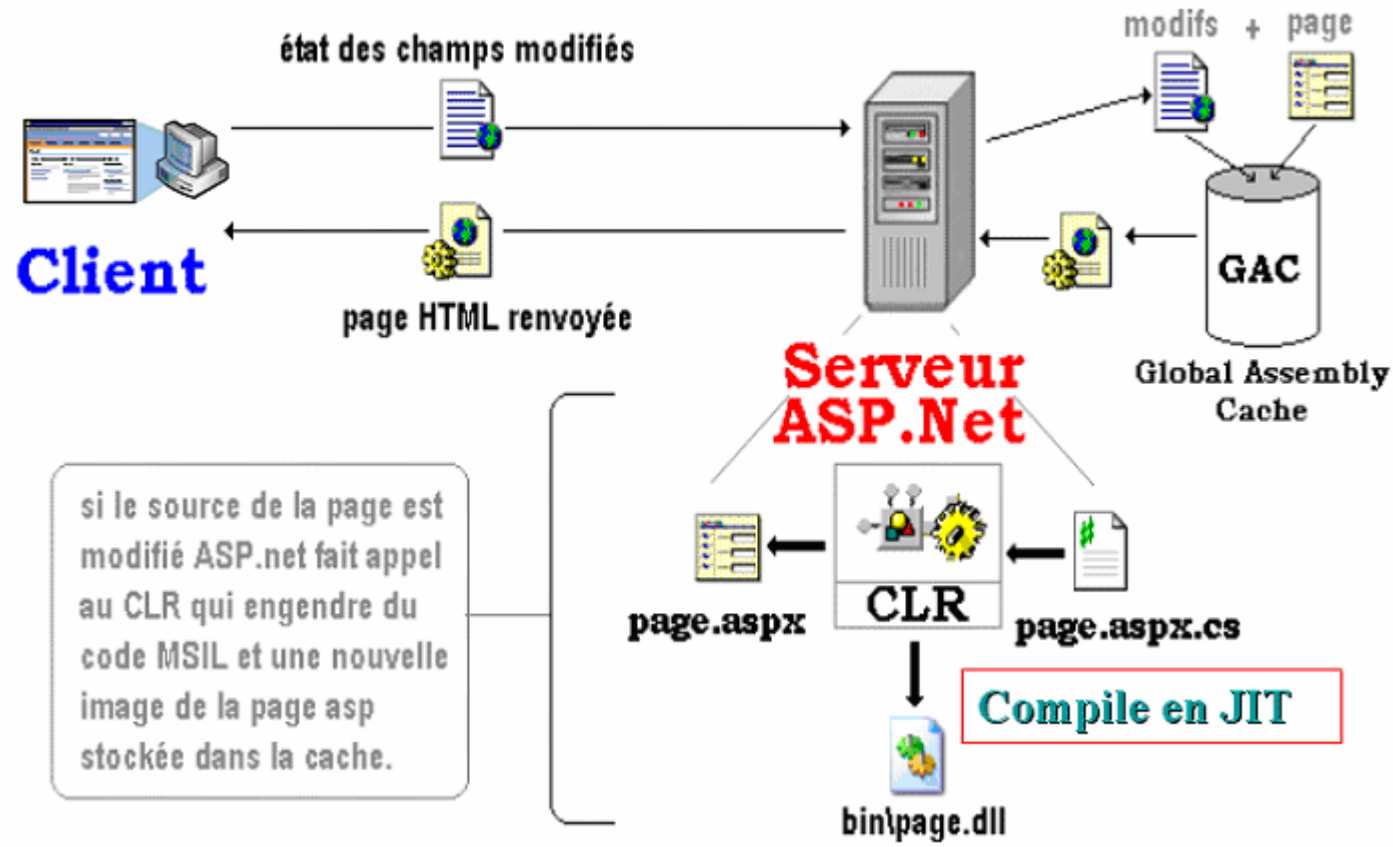
Code HTML engendré pour la page précédente et envoyé au client :

```
<head>
<title> Untitled Page </title>
</head>
<body>
    <form name="form1" method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTQ2OTkzNDMyMWRkU8v8pN7KXptqdUtkKvrRf6V6R1A=" />
</div>
    <div>
        <input type="submit" name="Button1" value="Button" id="Button1" />
    </div>
<div>
      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAgKUgJGxCwKM54rGBkvwYpMSUXqF04dObV7v1BzrERyr" />
</div></form>
</body>
```

Le code HTML envoyé au navigateur client est identique à celui qui a déjà été envoyé dans le cas déjà envisagé du bouton sans réaction événementielle.

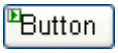
En effet, c'est le serveur ASP.Net qui maintient des fichiers de gestion des événements de la **Page** dans le fichier Default.aspx.cs et qui les exécute à partir de la mémoire cache du serveur selon les requêtes de l'utilisateur puis finalement renvoie une page HTML au navigateur client qui portera le nom Default.aspx.

Cette page HTML Default.aspx contient la réponse du serveur à la requête faite par le client.



4°) Les événements gérés côté serveur sont dans un script inclus (code-inline) :

Soit le même contrôle de classe **System.Web.UI.WebControls.Button** :



Au lieu de gérer sa réaction au click dans une page séparée, ASP.Net permet d'utiliser un deuxième modèle d'écriture le script inclus dans le code ASP.Net de la page (tous les langages supportés par .Net sont autorisés) :

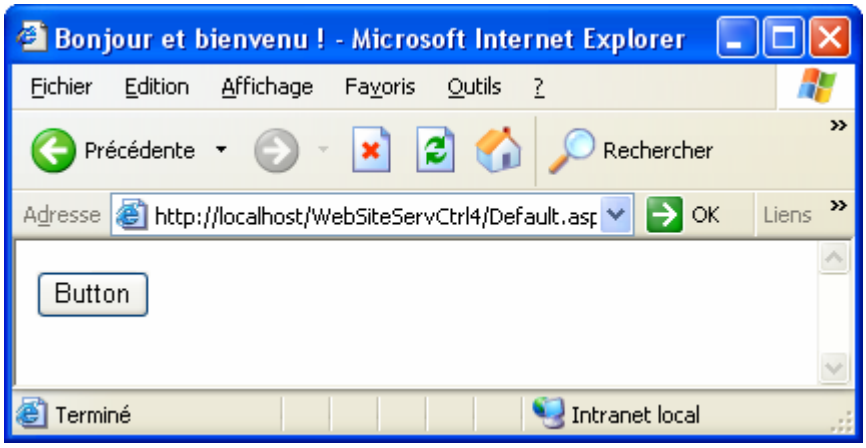
```
<%@ Page Language="C#"  %>
```

```
<script runat="server">
  protected void Button1_Click(object sender,
EventArgs e)
  {
    this.Title = "Bonjour et bienvenu !" ;
  }
</script>
```

```
<html>
<head runat="server">
    <title>Untitled Page</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
      <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
Text="Button" />
</div>
    </form>
</body>
</html>
```

Lorsque l'utilisateur clique sur le bouton, après envoie de la requête le serveur exécute le script du gestionnaire de l'événement click, comme il le fait si ce gestionnaire est situé dans une page à part (ici, la réponse consiste à modifier la barre de titre du navigateur en y mettant un message de bienvenue) :



Le panachage entre les deux modèles **code-inline** et **code-behind** est permis pour le même contrôle ou pour des contrôles différents.

Soit par exemple à programmer la réaction du bouton à l'événement click de souris grâce à un script C# et l'événement command du bouton avec du code séparé.

Code Asp.Net de la page Default.aspx construite :

```
<%@ Page Language="C#" AutoEventWireup="true"
                          CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<script runat="server">
  protected void Button1_Click(object sender, EventArgs e)
  {
    this.Title = "Bonjour et bienvenu !" ;
  }
</script>

<html>
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
      <asp:Button ID="Button1" runat="server"
                  OnClick="Button1_Click"
                  OnCommand="Button1_Command" Text="Button" />
</div>
    </form>
</body>
```

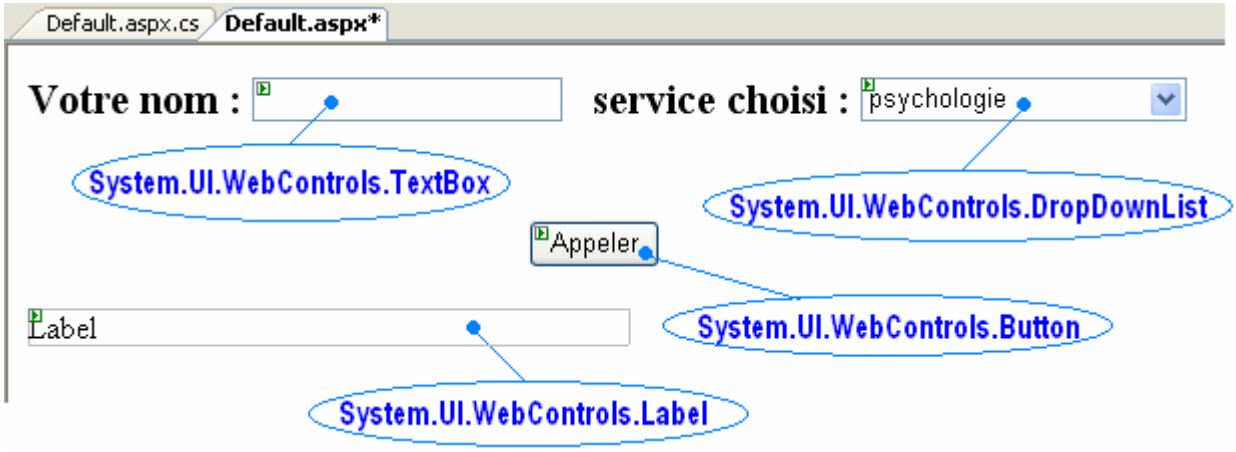
```
</html>
```

Code C# de la page Default.aspx.cs associée :

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void Button1_Command(object sender,
CommandEventArgs e)
    {
        Button1.Text = "Merci de votre visite";
    }
}
```

5°) Les événements côté serveur permettent des communications entre les contrôles :

Soit la page web suivante construite avec 4 contrôles web s'exécutant côté serveur :



Lorsque l'utilisateur clique dans sa page web sur le bouton [Appeler], le serveur exécute le code du gestionnaire de l'événement click programmé ci-dessous :

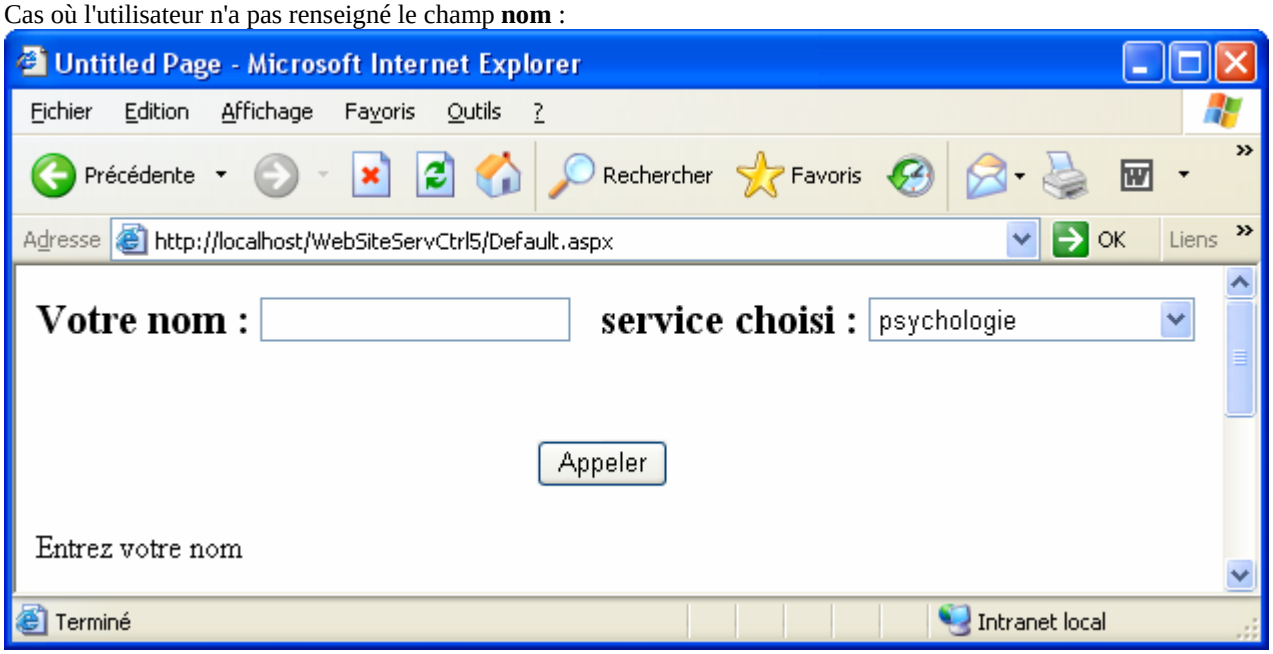
Code C# de la classe _Default dans la page Default.aspx.cs associée :

```
public partial class _Default : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
```

```
    {
        if (DropDownList1.SelectedValue != "" & TextBox1.Text
!= "")
        {
            Label1.Text = "Bonjour Mr. " + TextBox1.Text +
                           ", vous avez appelé le service de "
+

DropDownList1.SelectedValue;
        }
        else
            if (DropDownList1.SelectedValue == "")
                Label1.Text = "Choisissez un service";
            else
                if (TextBox1.Text == "")
                    Label1.Text = "Entrez votre nom";
    }
}
```

Affichages dans le navigateur client :

Cas où l'utilisateur n'a pas renseigné le champ **nom** :

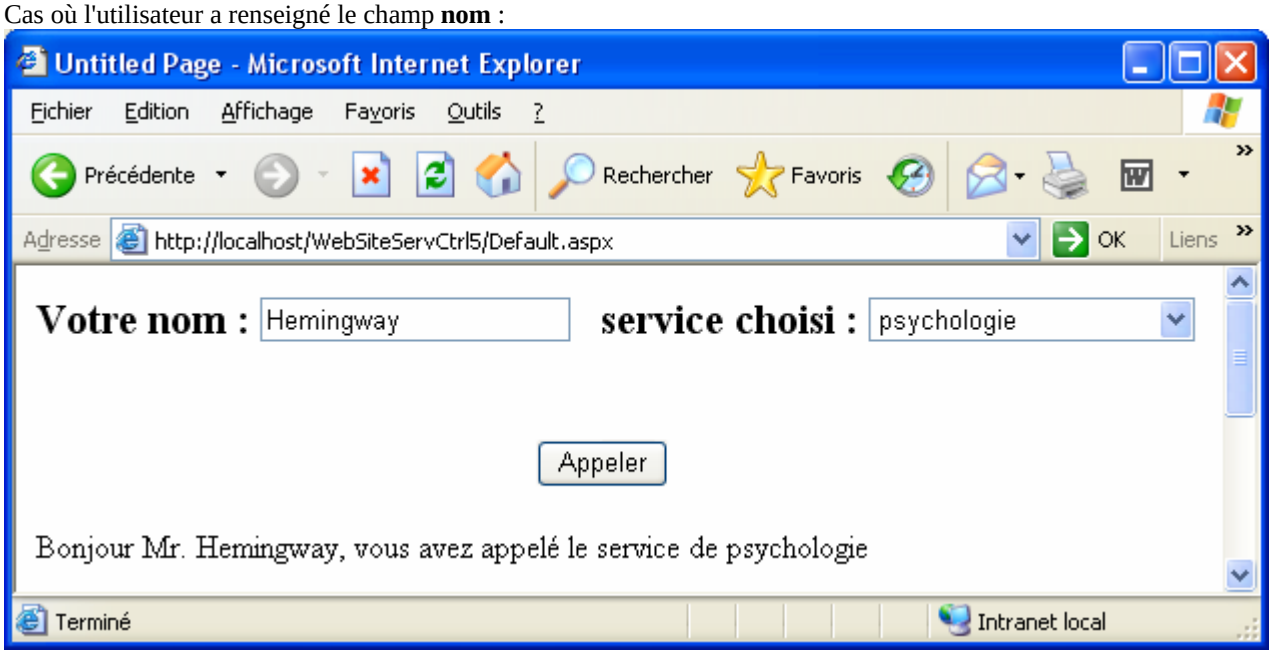


Code HTML engendré pour la page précédente et envoyé au client :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
      Untitled Page
</title></head>
<body>
    <form name="form1" method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTc4ODg2NDQzNw9kFgICAw9kFgICBw8PFgIeBFRleHQFEEVudHJleiB2b3RyZS
Bub21kZGTOfO+PtyHJUZDNUd4h+kXBkiTvpw==" />
```

```
</div>
<div id="DIV1">
  <span style="font-size: 16pt"><strong>Votre nom :
      <input name="TextBox1" type="text" id="TextBox1" />
         service choisi :
      <select name="DropDownList1" id="DropDownList1" style="width:163px;">
        <option selected="selected" value="psychologie">psychologie</option>
        <option value="pédiatrie">p&#233;diatrie</option>
        <option value="dermatologie">dermatologie</option>
     </select><br /><br /><br />
                     
                     
     
 <input type="submit" name="Button1" value="Appeler" id="Button1" />
</strong>
</span><br /><br />
<span id="Label1" style="display:inline-block;width:435px;">Entrez votre nom
</span><br /><br /><br /><br /><br /><br /><br /><br /><br /><br />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWBgL9w6GfDQLs0bLrBgLpo+XtBwL0goOiCQLfi9rZCAKM54rGBrH1QhYvJ/0kbCf0Fj
QcFvZhlM0a" />
</div>
</form>
</body>
</html>
```

Cas où l'utilisateur a renseigné le champ **nom** :



Code HTML engendré pour la page précédente et envoyé au client :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
      Untitled Page
</title></head>
<body>
    <form name="form1" method="post" action="Default.aspx" id="form1">
<div>
```

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTc4ODg2NDQzNw9kFgICAw9kFgICBw8PFgIeBFRleHQFEEVudHJleiB2b3RyZS
Bub21kZGTOfO+PtyHJUZDNUd4h+kXBkiTvpw==" />
</div>
<div id="DIV1">
  <span style="font-size: 16pt"><strong>Votre nom :
      <input name="TextBox1" type="text" value="Hemingway" id="TextBox1" />
         service choisi :
      <select name="DropDownList1" id="DropDownList1" style="width:163px;">
        <option selected="selected" value="psychologie">psychologie</option>
        <option value="pédiatrie">p&#233;diatrie</option>
        <option value="dermatologie">dermatologie</option>
     </select><br /><br /><br />
                     
                     
     
 <input type="submit" name="Button1" value="Appeler" id="Button1" />
</strong>
</span><br /><br />
<span id="Label1" style="display:inline-block;width:435px;">Bonjour Mr.
Hemingway, vous avez appelé le service de psychologie
</span>
<br /><br /><br /><br /><br /><br /><br /><br /><br /><br />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWBgL9w6GfDQLs0bLrBgLpo+XtBwL0goOiCQLfi9rZCAKM54rGBrH1QhYvJ/0kbCf0Fj
QcFvZhlM0a" />
</div>
</form>
</body>
</html>
```

Pour le debugging on trouve les fichiers xxx.dll et xxx.cs dans :

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files\websiteservctrl2\f6cd9c41\1150c7b3

Communication entre pages web

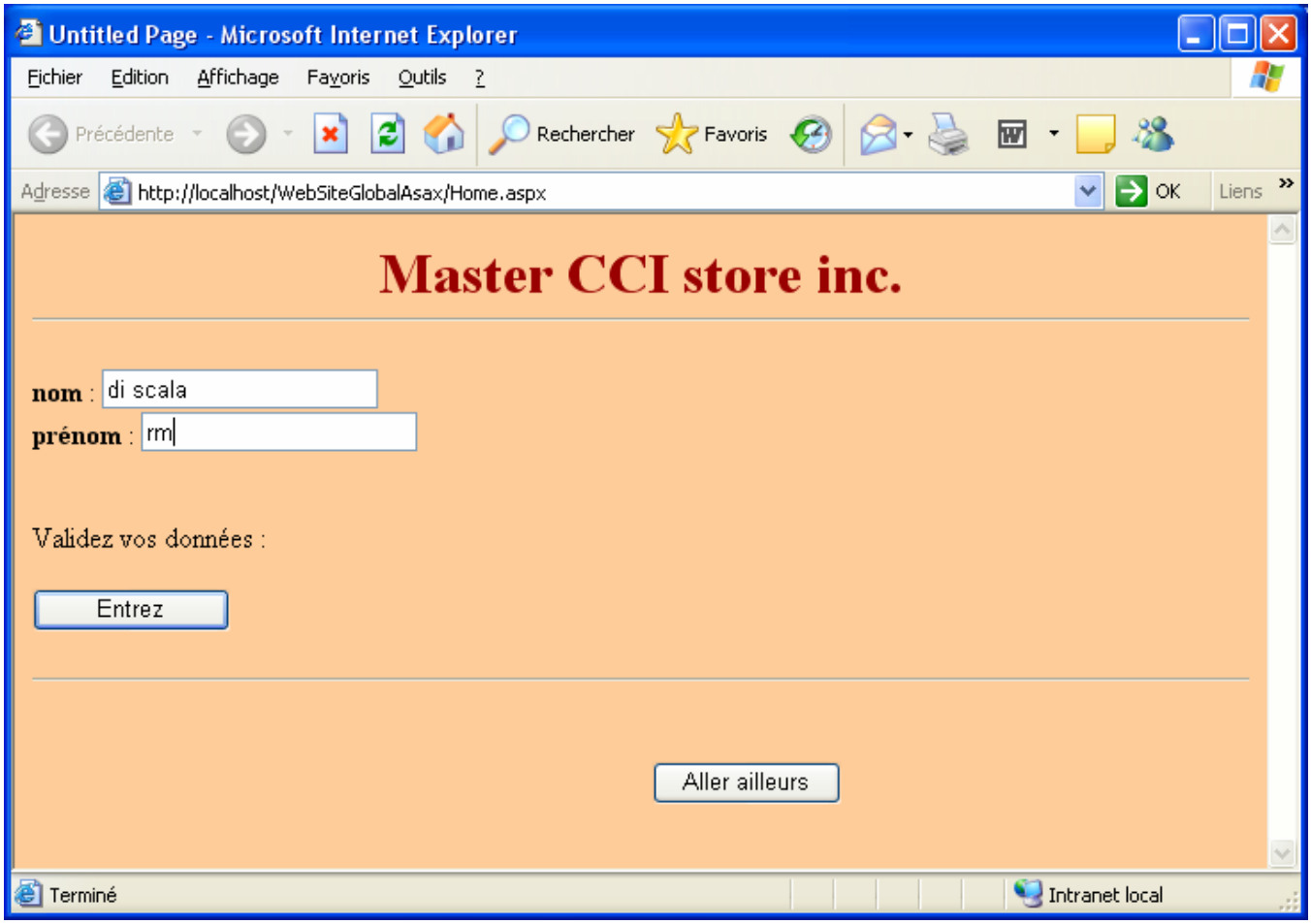
Plan général:

Objets utilisé pour la communication entre pages

1°) Objet Session
2°) Fichier Global.asax
3°) Répertoire App_Code

Communication entre pages web avec : l'objet de Session, le fichier Global.asax et le répertoire App_Code

Exemple d'utilisation avec deux pages :



Page Home.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Home.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body bgcolor="#ffcc99">
   <form id="form1" runat="server">
   <div style="text-align: center">
      <span style="font-size: 24pt; color: #6699ff"><strong><span style="color: #990000">Master
         CCI store inc.</span><br />
      </strong></span>
      <hr />
```

```
    </div>
      <br />
      <strong>nom</strong> :
      <asp:TextBox ID="TextBoxNom" runat="server"></asp:TextBox><br />
      <strong>prénom </strong>:
      <asp:TextBox ID="TextBoxPrenom" runat="server"></asp:TextBox><br />
      <br />
      <br />
      Validez vos données :<br />
      <br />
      <asp:Button ID="ButtonValider" runat="server" OnClick="ButtonValider_Click" Text="Entrez"
        Width="111px" /><br />
      <br />
      <hr />
      <br />
      <br />
                             
                             
                             
                    
      <asp:Button ID="ButtonEndSession" runat="server" OnClick="ButtonEndSession_Click"
        Text="Aller ailleurs" /><br />
       
    </form>
</body>
</html>
```

Page Home.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if ( !IsPostBack ) // premier chargement de cette page
        {
            this.Page.Session.Timeout = 1; //fin de la session au bout de 1mn sans requête
            if ((string)Page.Session["Nom"] != "...")
            {
                TextBoxNom.Text = (string)Page.Session["Nom"];
                TextBoxPrenom.Text = (string)Page.Session["Prenom"];
            }
        }
    }

    protected void ButtonValider_Click(object sender, EventArgs e)
    {
        string nom = TextBoxNom.Text;
        string prenom = TextBoxPrenom.Text;

        if (nom.Length != 0 & prenom.Length != 0)
```

```
        {
            this.Page.Session["Nom"] = nom;
            this.Page.Session["Prenom"] = prenom;
            Response.Redirect("Client.aspx");
        }
    }
    protected void ButtonEndSession_Click(object sender, EventArgs e)
    {
        this.Page.Session.Abandon();
        Response.Redirect("http://www.discala.net/");
    }
}
```

Fichier Global.asax

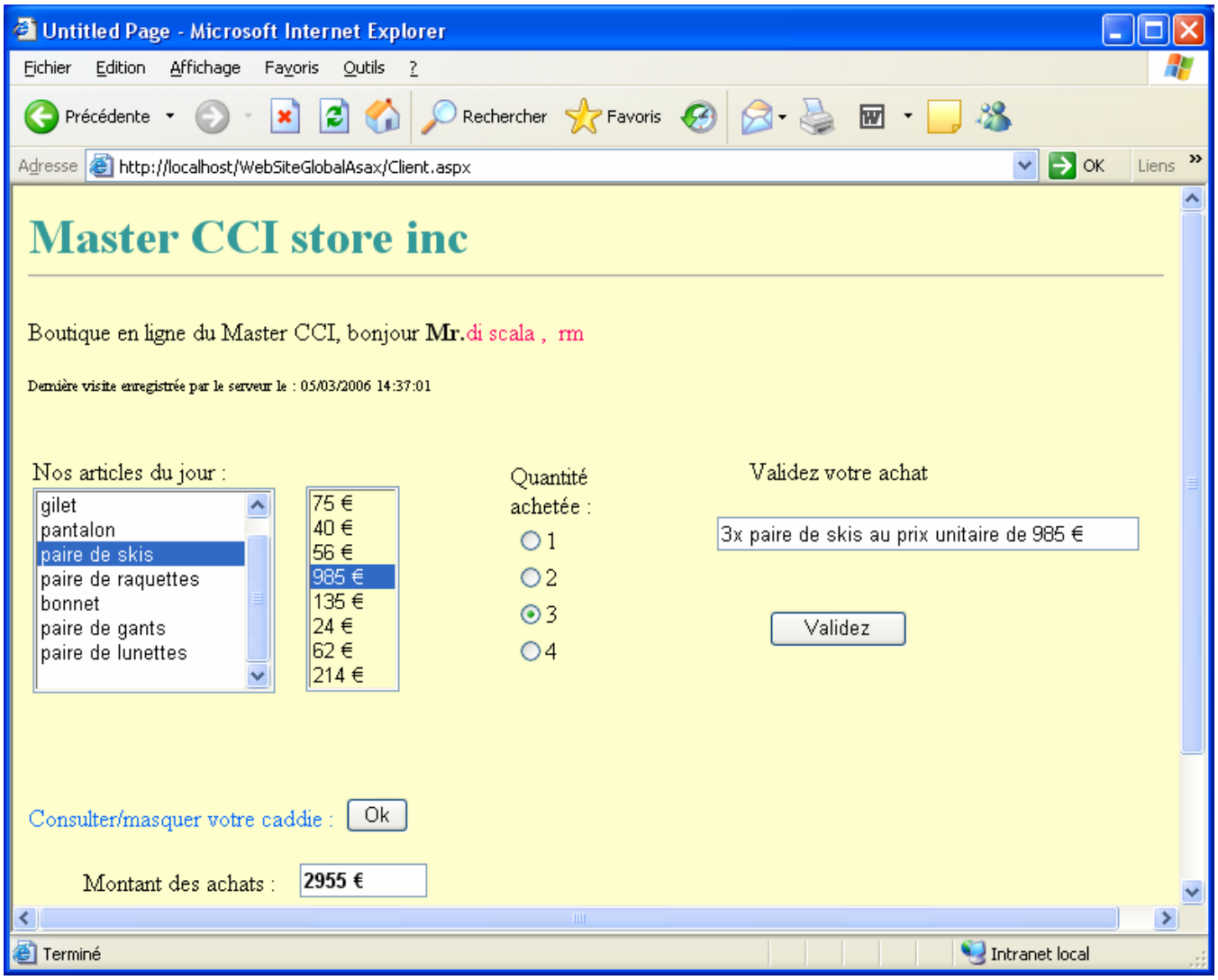
```
<%@ Application Language="C#" %>

<script RunAt="server">


    void Application_Start(object sender, EventArgs e)
    {
        // Code that runs on application startup
    }

    void Application_End(object sender, EventArgs e)
    {
        //  Code that runs on application shutdown
    }

    void Application_Error(object sender, EventArgs e)
    {
        // Code that runs when an unhandled error occurs
    }

    void Session_Start(object sender, EventArgs e)
    {
        // Code that runs when a new session is started
        this.Session["montant"] = 0d;
        this.Session["Nom"] = "...";
        this.Session["Prenom"] = "...";
        Class1.LireFichier();
    }

    void Session_End(object sender, EventArgs e)
    {
        // Code that runs when a session ends.
        // Note: The Session_End event is raised only when the sessionstate mode
        // is set to InProc in the Web.config file. If session mode is set to StateServer
        // or SQLServer, the event is not raised.
        Class1.ecrireFichier((string)Session["Nom"], (string)Session["Prenom"]);
    }

</script>
```

Code dans App_Code : Class1.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Collections.Generic;
using System.IO;

/// <summary>
/// Class1 dans le dossier App_Code (version ASP.net 2.0)
/// </summary>
public class Class1
{
   public static List<string> listeClients = new List<string>();
   private static string fichierClients = @"C:\Inetpub\Datas\clients.txt";

   public Class1()
   {
      //
      // TODO: Add constructor logic here
      //
   }
   public static void LireFichier()
   {
      if (File.Exists(fichierClients))
      {
         using (StreamReader fluxRead = new StreamReader(fichierClients))
         {
            string ligne;
            listeClients.Clear();
            while ((ligne = fluxRead.ReadLine()) != null)
               listeClients.Add(ligne);
         }
      }
   }
   public static void ecrireFichier(string nom, string prenom)
   {
      if (nom.Length != 0 & prenom.Length != 0)
         if (File.Exists(fichierClients))
         {
            using (StreamWriter fluxWrite = new StreamWriter(fichierClients, true))
            {
               fluxWrite.WriteLine(nom + "*" + prenom + "*" + DateTime.Now);
            }
         }
         else
            using (StreamWriter fluxWrite = new StreamWriter(fichierClients))
            {
               fluxWrite.WriteLine(nom + "*" + prenom + "*" + DateTime.Now);
            }
   }
}
```

Il est maintenu en accès **static** une recopie en MC du fichier des clients déjà passés sur le site "c:\Inetpub\Datas\clients.txt", lors de l'ouverture d'une session d'un nouvel utilisateur : public static List<string> listeClients (chargement de la liste, sur l'événement Session_Start géré dans Global.asax).

Lors de validation des données d'une personne s'étant identifiée, le serveur consulte la liste des clients déjà venus et affiche éventuellement dans la nouvelle page web Client.aspx, un rappel de la date de la dernière visite effectuée par cette personne :



Sur cette page Client.aspx, l'utilisateur peut sélectionner avec la souris dans une liste, un article dont le prix est automatiquement affiché dans une autre liste. L'utilisateur dispose de la possibilité d'acheter de 1 à 4 articles du même type, le serveur affiche dès validation du choix le montant des achats en cours.

Page Client.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Client.aspx.cs" Inherits="Default2" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
```

```
<head runat="server">
   <title>Untitled Page</title>
</head>
<body bgcolor="#ffffcc">
   <form id="form1" runat="server">
   <div>
      <span style="font-size: 24pt; color: #339999"><strong>Master CCI store inc</strong></span><br />
      <hr />
      <br />
      Boutique en ligne du Master CCI, bonjour <span style="color: #ff0066"><strong><span
         style="color: #000000"> Mr.</span></strong><%=this.Page.Session["Nom"] %> , 
         <%= this.Page.Session["Prenom"]%></span><br />
      <br />
      <asp:Label ID="LabelDejaVenu" runat="server" Font-Size="X-Small" Visible="False"
         Width="611px"></asp:Label><br />
      <table style="width: 763px; height: 133px">
         <tr>
            <td style="width: 271px; height: 171px">
      Nos articles du jour :<br />
      <asp:ListBox ID="ListBoxArticle" runat="server" Height="141px" Width="158px" AutoPostBack="True"

OnSelectedIndexChanged="ListBoxArticle_SelectedIndexChanged">
         <asp:ListItem Selected="True">paire de chaussures</asp:ListItem>
         <asp:ListItem>gilet</asp:ListItem>
         <asp:ListItem>pantalon</asp:ListItem>
         <asp:ListItem>paire de skis</asp:ListItem>
         <asp:ListItem>paire de raquettes</asp:ListItem>
         <asp:ListItem>bonnet</asp:ListItem>
         <asp:ListItem>paire de gants</asp:ListItem>
         <asp:ListItem>paire de lunettes</asp:ListItem>
         <asp:ListItem></asp:ListItem>
      </asp:ListBox>    
      <asp:ListBox ID="ListBoxPrix" runat="server" Height="142px" Width="61px" BackColor="#FFFFCC">
         <asp:ListItem Value="75,0" Selected="True">75 €</asp:ListItem>
         <asp:ListItem Value="40,0">40 €</asp:ListItem>
         <asp:ListItem Value="56,0">56 €</asp:ListItem>
         <asp:ListItem Value="985,0">985 €</asp:ListItem>
         <asp:ListItem Value="135,0">135 €</asp:ListItem>
         <asp:ListItem Value="24,0">24 €</asp:ListItem>
         <asp:ListItem Value="62,0">62 €</asp:ListItem>
         <asp:ListItem Value="214,0">214 €</asp:ListItem>
      </asp:ListBox></td>
            <td style="width: 89px; height: 171px">
      Quantité achetée :    
               <br />
      <asp:RadioButtonList id="RadioButtonListChoix" runat="server" Width="116px"
            OnSelectedIndexChanged="RadioButtonListChoix_SelectedIndexChanged" AutoPostBack="True">
            <asp:ListItem>1</asp:ListItem>
            <asp:ListItem>2</asp:ListItem>
            <asp:ListItem>3</asp:ListItem>
            <asp:ListItem>4</asp:ListItem>
      </asp:RadioButtonList><br />
            </td>
            <td style="width: 159px; height: 171px; text-align: center">
               <br />
               <br />
               Validez votre achat<br />
               <br />
               <asp:TextBox ID="TextBoxAchat" runat="server" ReadOnly="True" Width="269px"></asp:TextBox>
               <br />
```

```
                <br />
                <asp:Button ID="ButtonValider" runat="server" Text="Validez" OnClick="ButtonValider_Click"
                                                                                                Width="90px" /><br />
                <br />
                <br />
                <br />
                <br />
            </td>
        </tr>
    </table>
    <br />
    <span style="color: #0066ff">Consulter/masquer
    votre caddie :</span> 
    <asp:Button ID="ButtonConsult" runat="server" Text="Ok" OnClick="ButtonConsult_Click" Width="41px"
Height="23px" /><br />
    <br />
    <asp:ListBox ID="ListBoxCaddie" runat="server" Height="142px" Width="293px" BackColor="Azure"
Visible="False"></asp:ListBox>
             Montant des achats :    <asp:TextBox
      ID="TextBoxMontant" runat="server" Font-Bold="True" ReadOnly="True"
Width="77px"></asp:TextBox>
                <br />
    <br />
    <br />
    <br />
    <asp:HyperLink ID="HyperLinkAccueil" runat="server" Font-Bold="True" NavigateUrl="~/Home.aspx"
      Width="139px">Revenir à l'accueil</asp:HyperLink><br />
    <br />
  </div>
  </form>
</body>
</html>
```

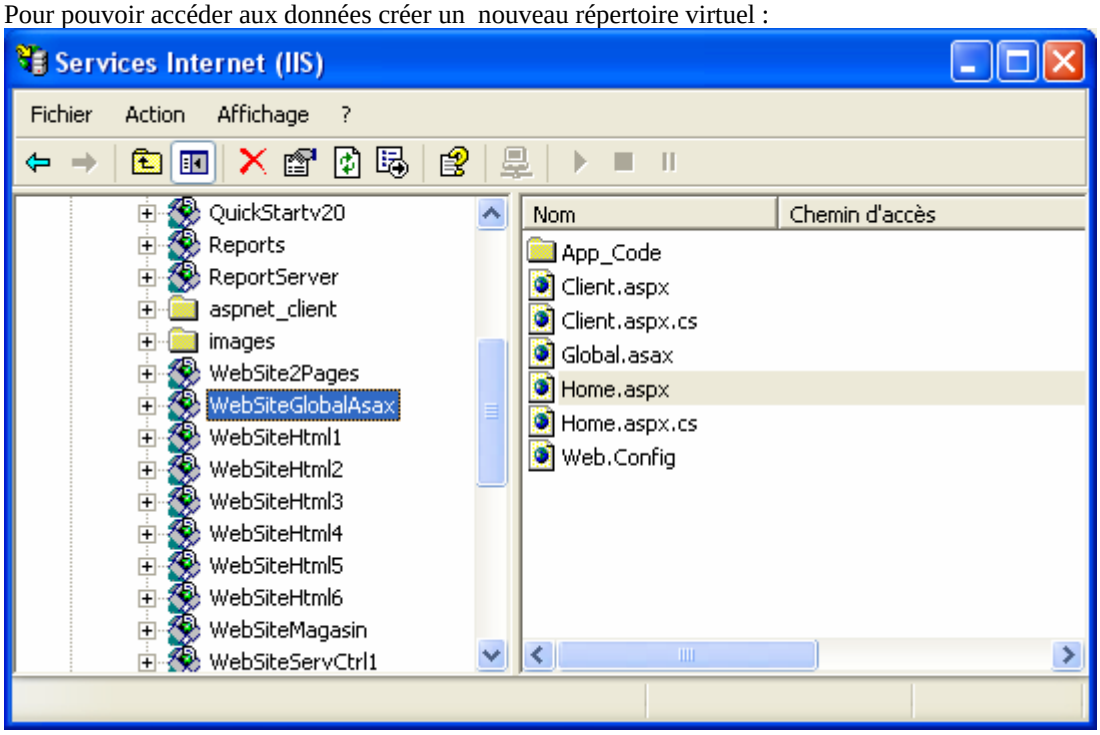
Page Client.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class Default2 : System.Web.UI.Page
{
  private double montant = 0d;

  protected void Page_Load(object sender, EventArgs e)
  {
    if (!IsPostBack) // premier chargement de cette page
    {
      LabelDejaVenu.Visible = searchIdentite((string)Page.Session["Nom"], (string)Page.Session["Prenom"]);
    }
    montant = (double)this.Page.Session["montant"];
  }
```

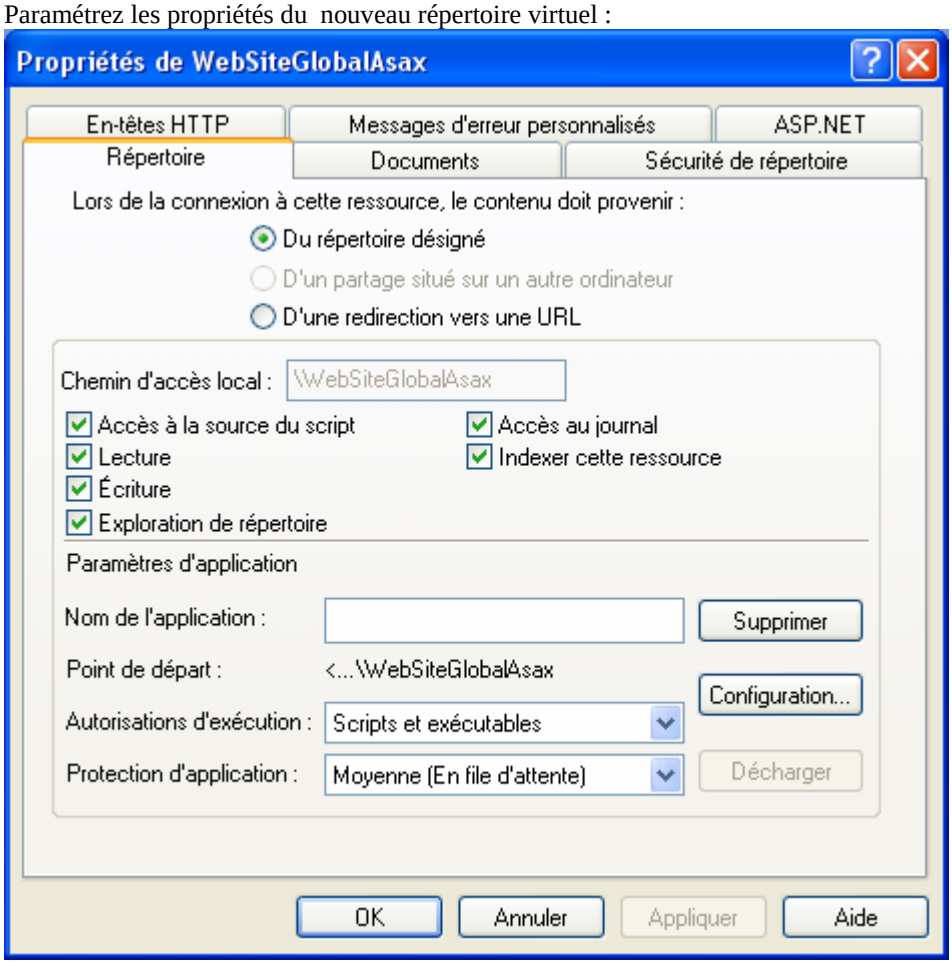
```
    private bool searchIdentite(string nom, string prenom)
    {
        bool dejaVenu = false;
        LabelDejaVenu.Text = "";
        foreach (string s in Class1.listeClients)
            if (s.IndexOf(nom) != -1 & s.IndexOf(prenom) !=-1)
            {
                LabelDejaVenu.Text = "Dernière visite enregistrée par le serveur le : " + s.Split('*')[2];
                dejaVenu = true;
            }
        return dejaVenu;
    }

    private void selectionShow()
    {
        ListBoxCaddie.Visible = false;
        if (RadioButtonListChoix.SelectedIndex == -1)
            RadioButtonListChoix.SelectedIndex = 0;
        this.TextBoxAchat.Text = RadioButtonListChoix.SelectedItem + "x " +
            ListBoxArticle.SelectedItem + " au prix unitaire de " + ListBoxPrix.SelectedItem;
    }
    protected void ListBoxArticle_SelectedIndexChanged(object sender, EventArgs e)
    {
        ListBoxPrix.SelectedIndex = ListBoxArticle.SelectedIndex;
        selectionShow();
    }

    protected void RadioButtonListChoix_SelectedIndexChanged(object sender, EventArgs e)
    {
        selectionShow();
    }

    protected void ButtonValider_Click(object sender, EventArgs e)
    {
        if (TextBoxAchat.Text.Length != 0)
        {
            selectionShow();
            ListBoxCaddie.Items.Add(TextBoxAchat.Text);
            montant += Convert.ToDouble(ListBoxPrix.SelectedValue) *

Convert.ToDouble(RadioButtonListChoix.SelectedValue);
            TextBoxMontant.Text = montant.ToString() + " €";
            this.Page.Session["montant"] = montant;
        }
    }

    protected void ButtonConsult_Click(object sender, EventArgs e)
    {
        if (TextBoxAchat.Text.Length != 0)
        {
            ListBoxCaddie.Visible = !ListBoxCaddie.Visible;
        }
    }
}
```

Serveur internet IIS

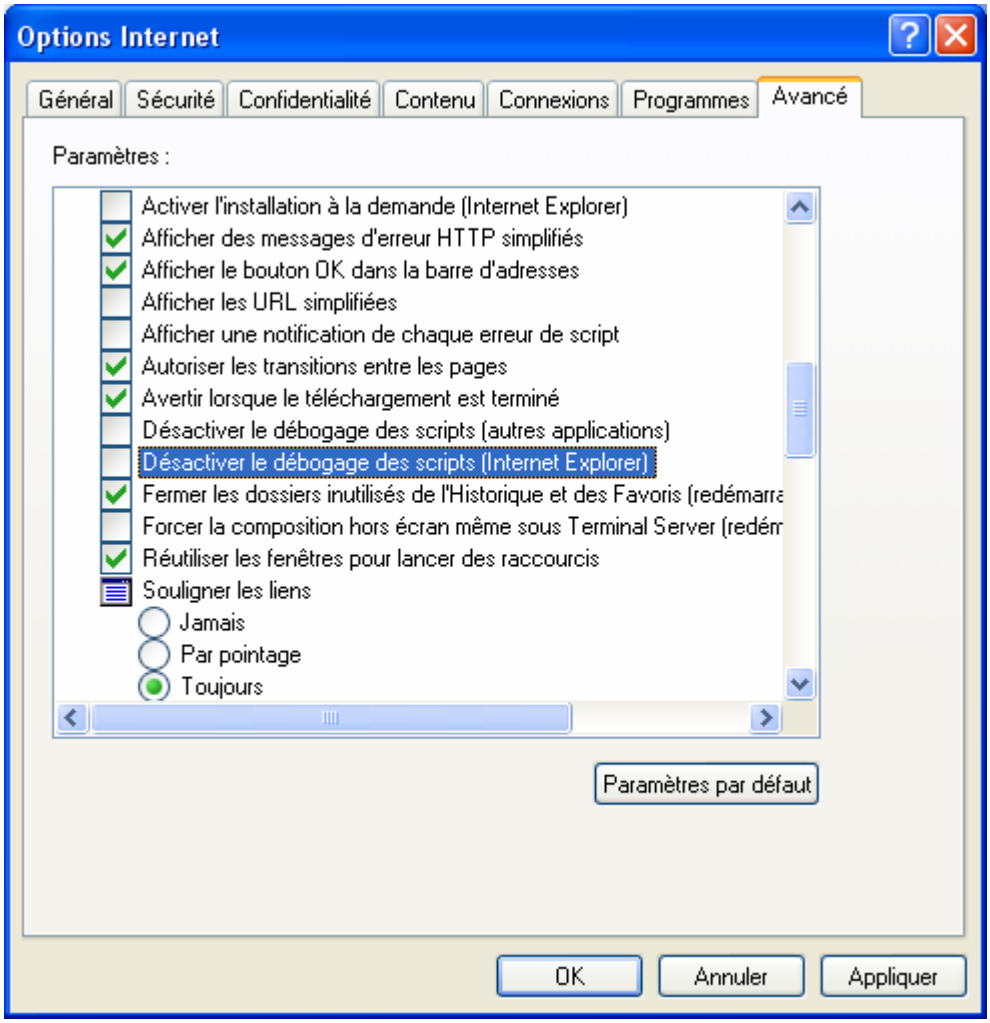
Pour pouvoir accéder aux données créer un nouveau répertoire virtuel :



Paramétrez les propriétés du nouveau répertoire virtuel :



Pour pouvoir debugger : dans le navigateur "Internet explorer IE5 et +" menu **outils\options internet**



Laisser le débogage activé si vous voulez débugger vos script dans vos programmes ASP.Net.

Le débogage de Visual studio sur le C# fonctionne quant à lui normalement, seuls les scripts C# (ou autre langage) ont besoin du paramétrage précédent du navigateur Internet.

ASP.net etBD avec un GridView

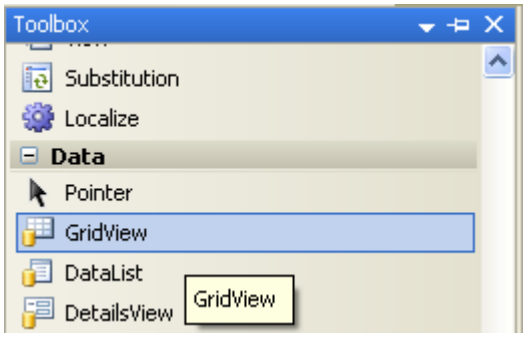
Plan général:

Comment utiliser un GridView pour accéder à une BD

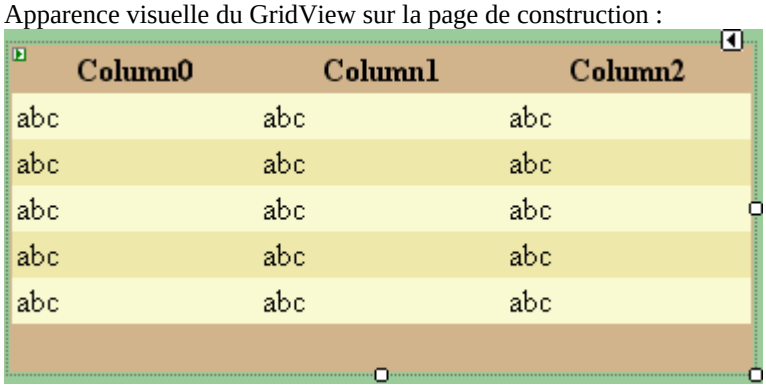
1°) Accès à une BD SqlServer à travers les nouveaux composants .net
2°) Accès à des données hiérarchiques avec un XmlDataSource
3°) Exemple d'utilisation d'un cookie.

ASP.NET et BD avec GridView

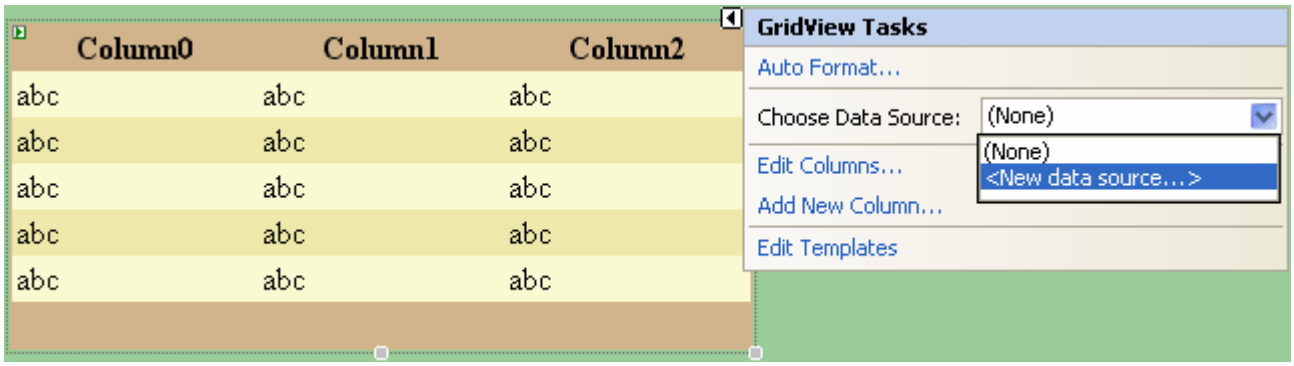
Le contrôle **GridView** du namespace **System.Web.UI.WebControls** permet d'afficher, d'éditer, et d'effacer des données principalement à partir d'une base de données (Sql server, Oracle, Access, …) et à travers le composant **SqlDataSource** :



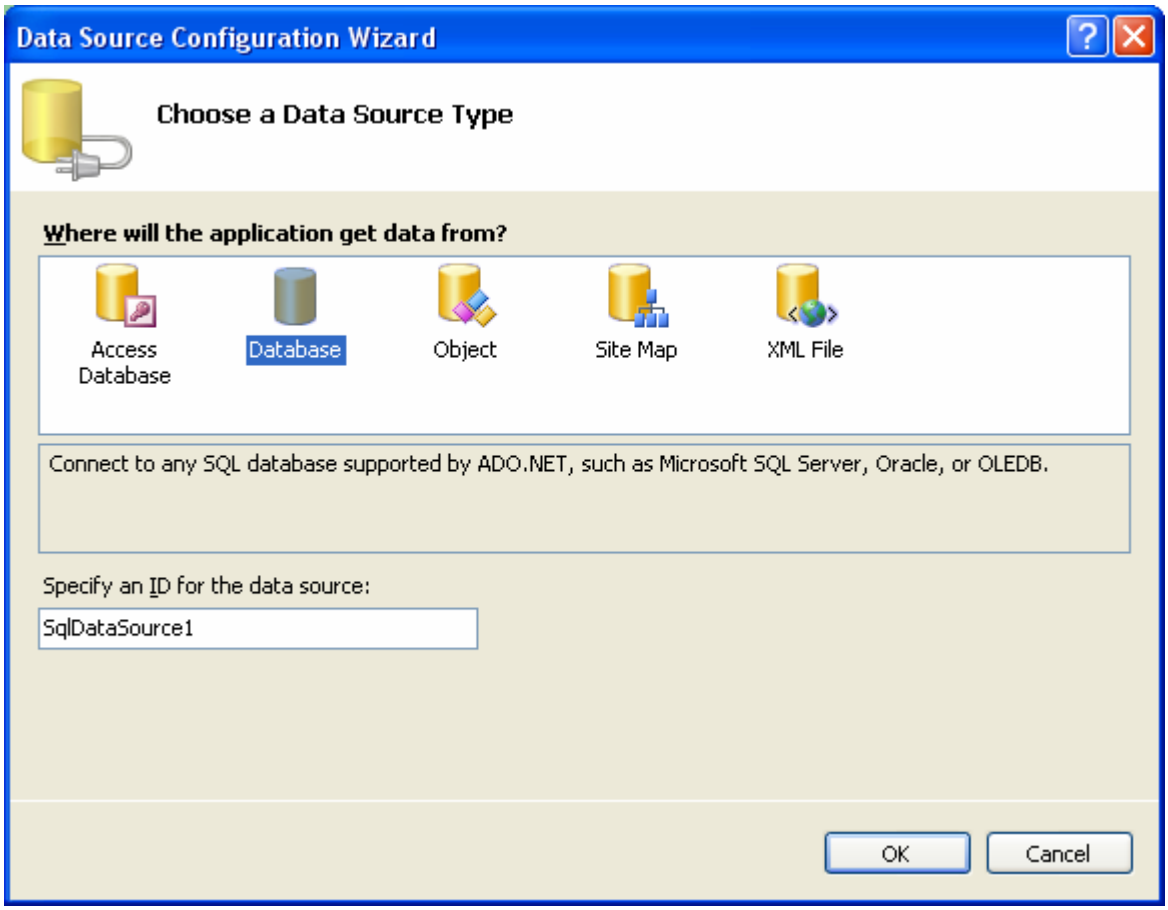
Apparence visuelle du GridView sur la page de construction :



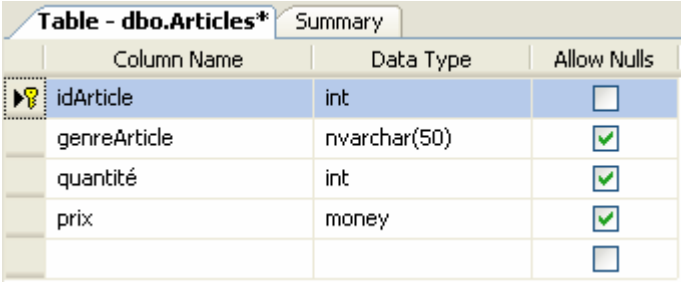
Demande de création d'une liaison avec une source de données :

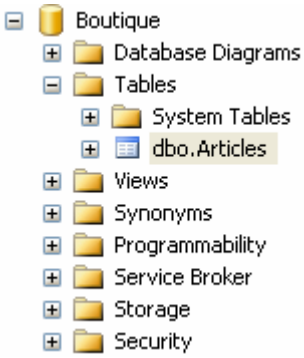


Parmi les sources de données possibles, comme nous voulons visualiser une table extraite d'une base de données (Sql server, Oracle,…) il nous faut utiliser le composant SqlDataSource que l'on sélectionne par l'icône Database dans l'expert de configuration :

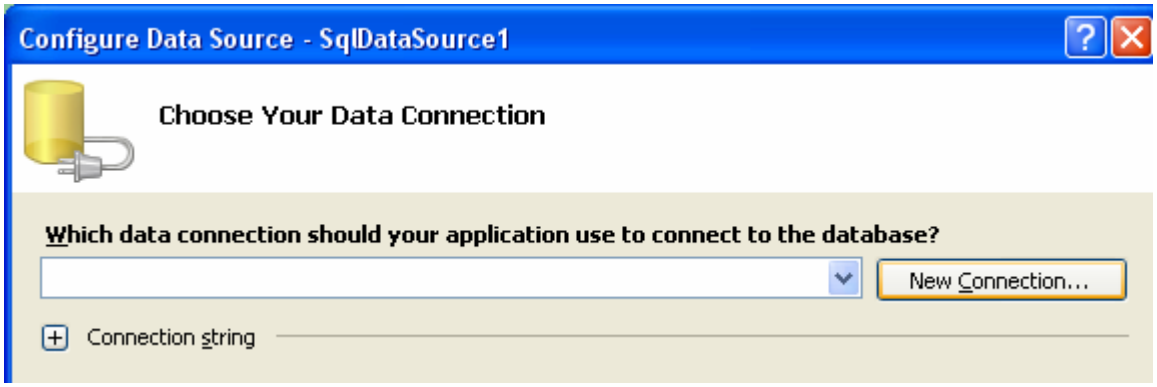


Ci-dessous la création de la base de données "Boutique.mdf"et de la table "Articles" avec Sql server management studio:

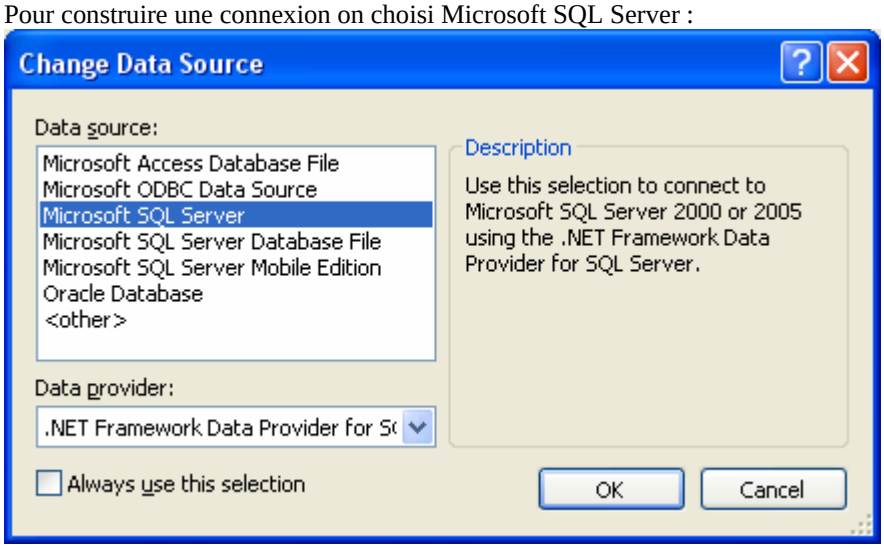




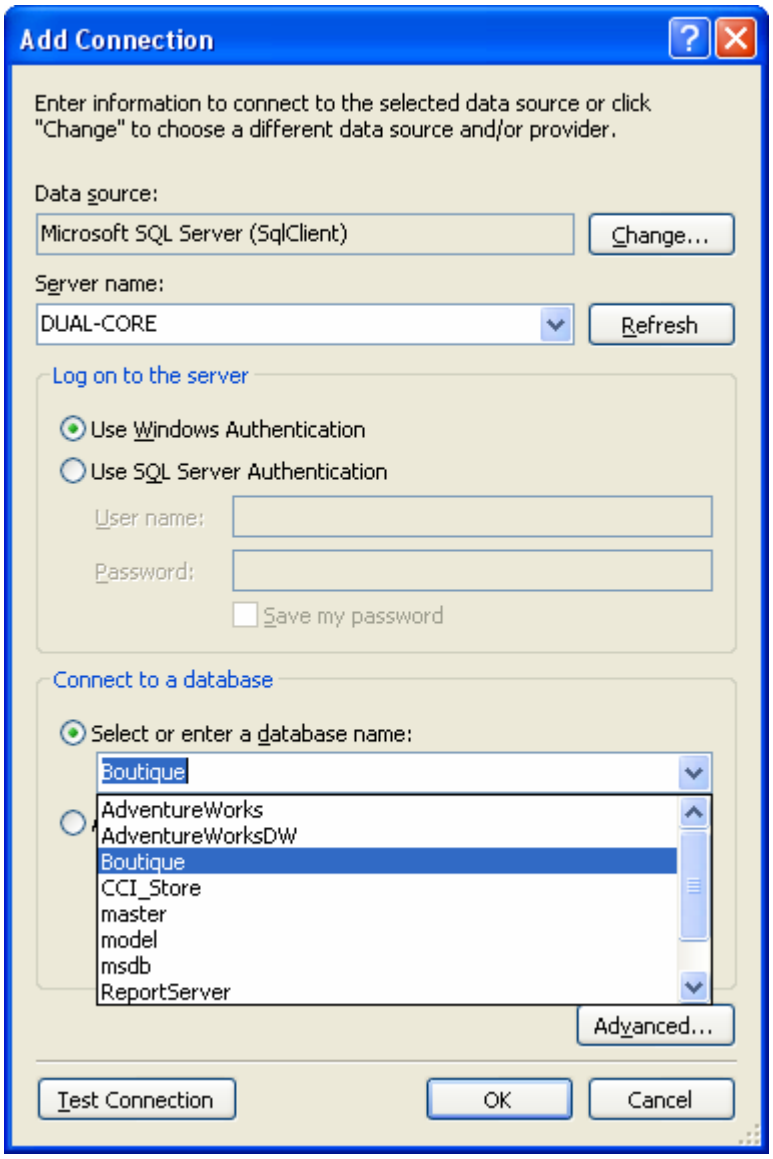
Les étapes de connexion de l'objet SqlDataSource à Sql Server 2005 :

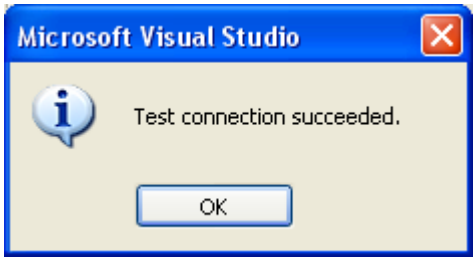


Pour construire une connexion on choisi Microsoft SQL Server :



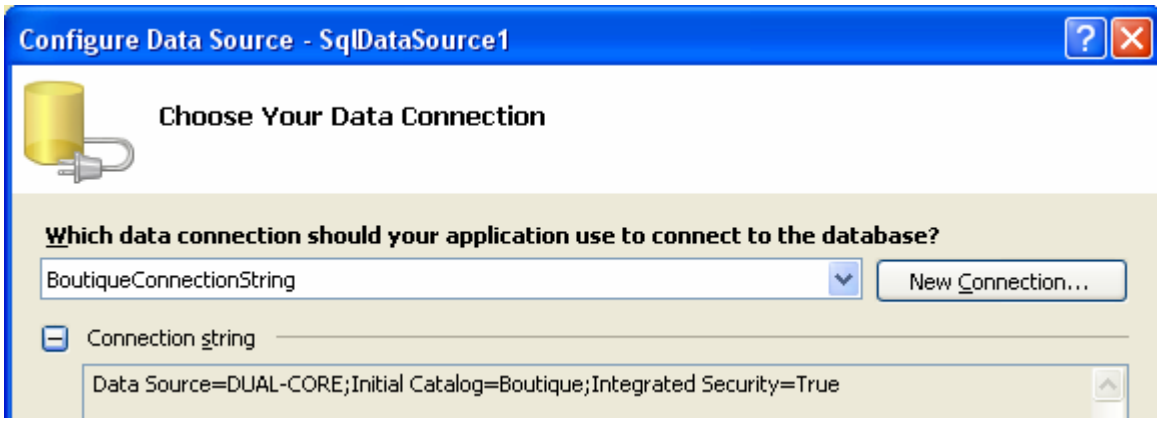
Il faut nommer le serveur ici (DUAL-CORE) et la BD (ici Boutique) :





(pour mémoire la BD "Boutique.mdf" a été rangée lors de sa création, dans MSSQL.1\MSSQL\Data\ )

Une string de connexion est créée à partir des renseignements qui viennent d'être fournis :

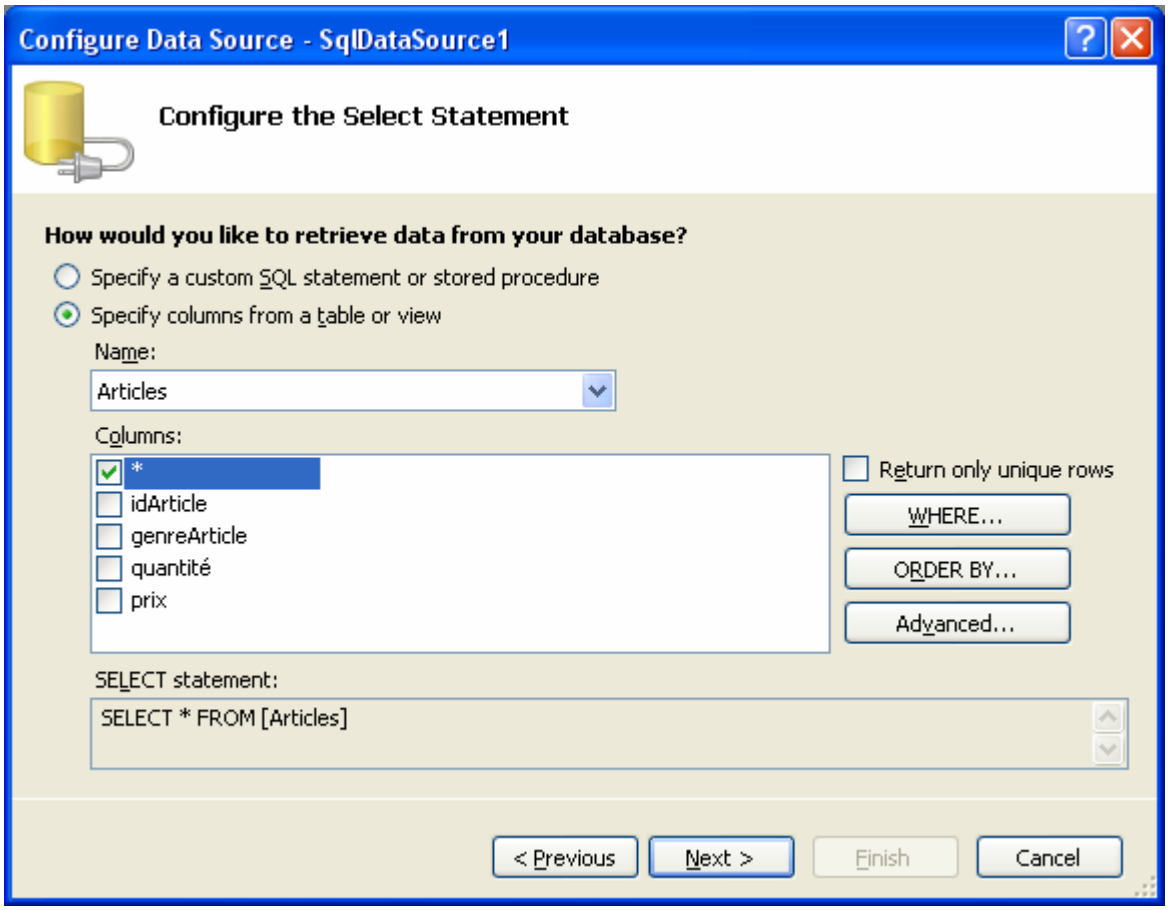


Cette chaîne de connexion est stockée dans le fichier **Web.config** sous le nom BoutiqueConnectionString

Dans Web.config :

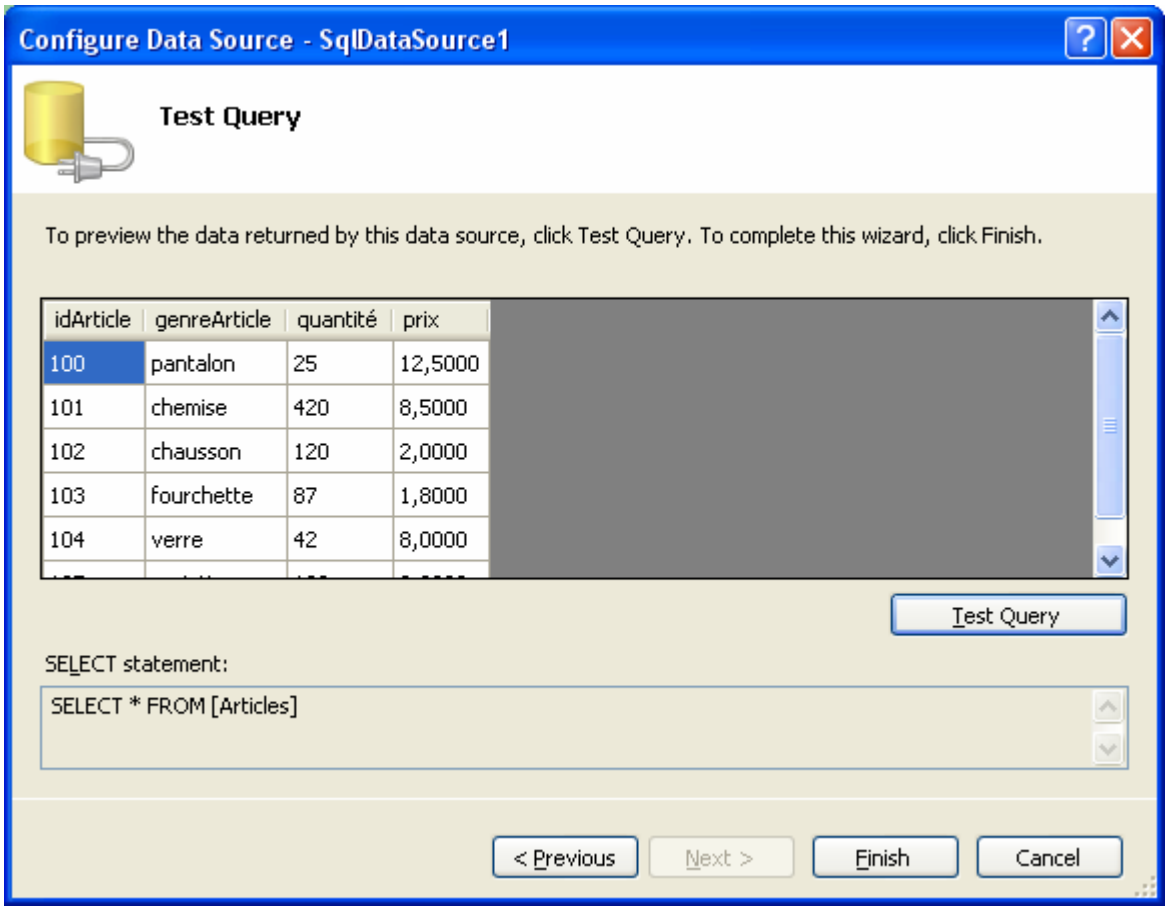
```
<connectionStrings>
  <add name="BoutiqueConnectionString"
      connectionString="Data Source=DUAL-CORE;
                        Initial Catalog=Boutique;
                        Integrated Security=True"
      providerName="System.Data.SqlClient" />
 </connectionStrings>
```

L'expert de construction permet ensuite de configurer l'envoi de la commande SELECT à la BD :

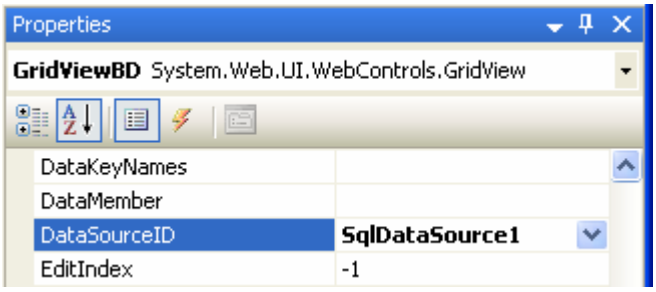


Ici cette commande est : SELECT * FROM [Articles]

L'expert de construction permet de visualiser à des fins de correction, le résultat de la commande choisie :



Le GridView est lié au SqlDataSource par sa propriété DataSourceID :

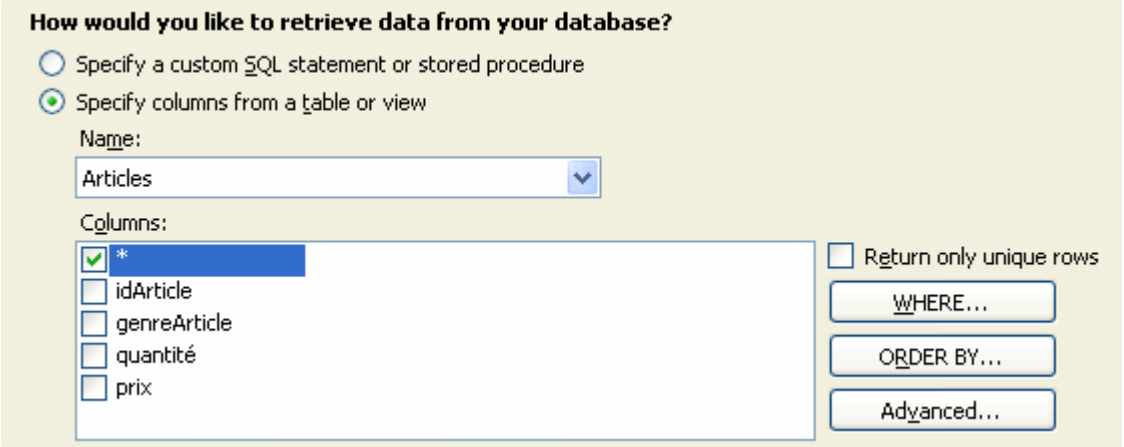


voici le code ASP.net engendré :

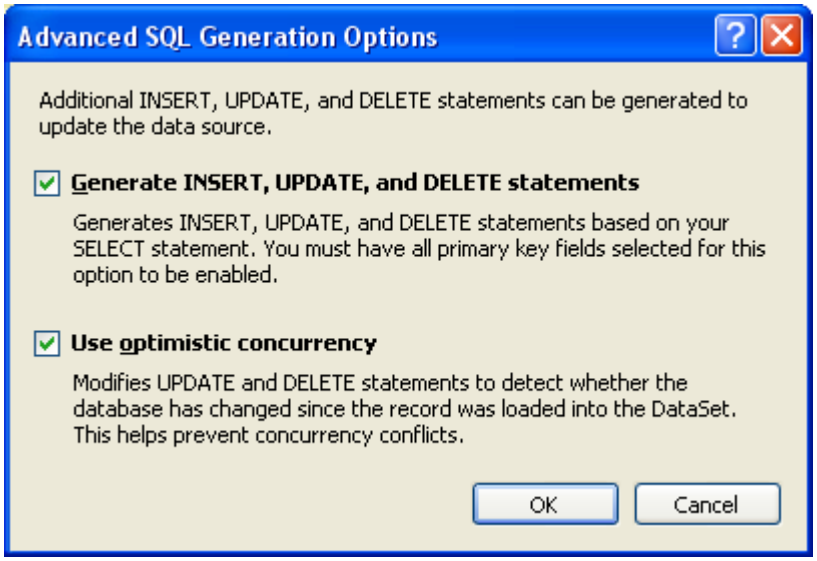
```
        <asp:GridView ID="GridViewBD" runat="server"
AutoGenerateColumns="False"
                BackColor="LightGoldenrodYellow"
                BorderColor="Tan" BorderWidth="1px" CellPadding="2"
                DataSourceID="SqlDataSource1" ForeColor="Black"
GridLines="None" Width="371px" RowHeaderColumn="idArticle" ShowFooter="True">
            <FooterStyle BackColor="Tan" />
            <SelectedRowStyle BackColor="DarkSlateBlue"
ForeColor="GhostWhite" />
            <PagerStyle BackColor="PaleGoldenrod" ForeColor="DarkSlateBlue"
```

```
HorizontalAlign="Center" />
            <HeaderStyle BackColor="Tan" Font-Bold="True" />
            <AlternatingRowStyle BackColor="PaleGoldenrod" />
        </asp:GridView>

        <asp:SqlDataSource ID="SqlDataSource1" runat="server"
           ConnectionString="<%$ ConnectionStrings:BoutiqueConnectionString
%>"
           ProviderName="<%$
ConnectionStrings:BoutiqueConnectionString.ProviderName %>"
           SelectCommand="SELECT * FROM [Articles]">
        </asp:SqlDataSource>
```

Si l'on sélectionne le bouton Advanced generation options dans l'expert de construction :



on peut choisir de générer automatiquement les instructions INSERT, UPDATE et DELETE :

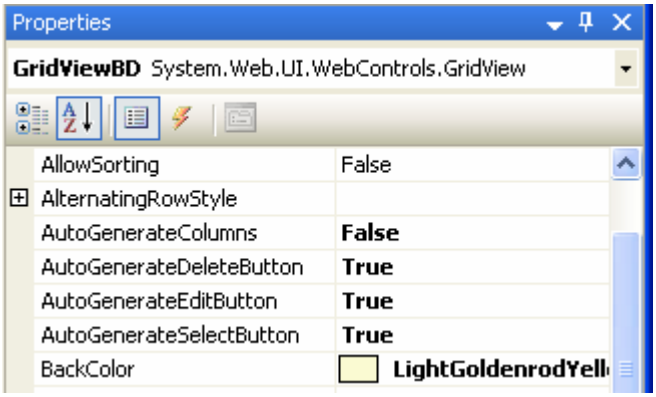


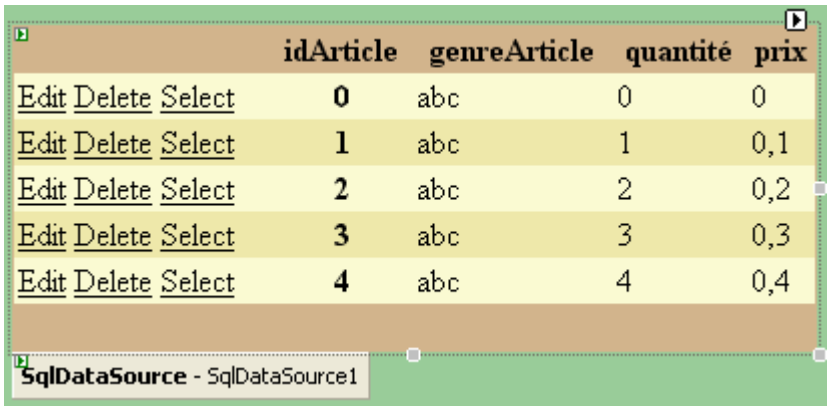
voici le code ASP.net engendré :

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
           ConnectionString="<%$ ConnectionStrings:BoutiqueConnectionString
%>"
           ProviderName="<%$
ConnectionStrings:BoutiqueConnectionString.ProviderName %>"
           SelectCommand="SELECT * FROM [Articles]"
           ConflictDetection="CompareAllValues"
```

```
            DeleteCommand="DELETE FROM [Articles] WHERE [idArticle] =
@original_idArticle AND [genreArticle] = @original_genreArticle AND
[quantité] = @original_quantité AND [prix] = @original_prix"
            InsertCommand="INSERT INTO [Articles] ([idArticle],
[genreArticle], [quantité], [prix]) VALUES (@idArticle, @genreArticle,
@quantité, @prix)"
            OldValuesParameterFormatString="original_{0}"
            UpdateCommand="UPDATE [Articles] SET [genreArticle] =
@genreArticle, [quantité] = @quantité, [prix] = @prix WHERE [idArticle] =
@original_idArticle AND [genreArticle] = @original_genreArticle AND
[quantité] = @original_quantité AND [prix] = @original_prix">
            <DeleteParameters>
                <asp:Parameter Name="original_idArticle" Type="Int32" />
                <asp:Parameter Name="original_genreArticle" Type="String" />
                <asp:Parameter Name="original_quantit&#233;" Type="Int32" />
                <asp:Parameter Name="original_prix" Type="Decimal" />
            </DeleteParameters>
            <UpdateParameters>
                <asp:Parameter Name="genreArticle" Type="String" />
                <asp:Parameter Name="quantit&#233;" Type="Int32" />
                <asp:Parameter Name="prix" Type="Decimal" />
                <asp:Parameter Name="original_idArticle" Type="Int32" />
                <asp:Parameter Name="original_genreArticle" Type="String" />
                <asp:Parameter Name="original_quantit&#233;" Type="Int32" />
                <asp:Parameter Name="original_prix" Type="Decimal" />
            </UpdateParameters>
            <InsertParameters>
                <asp:Parameter Name="idArticle" Type="Int32" />
                <asp:Parameter Name="genreArticle" Type="String" />
                <asp:Parameter Name="quantit&#233;" Type="Int32" />
                <asp:Parameter Name="prix" Type="Decimal" />
            </InsertParameters>
        </asp:SqlDataSource>
```

Paramétrer les boutons d'édition, de suppression et de sélection dans le GridView :





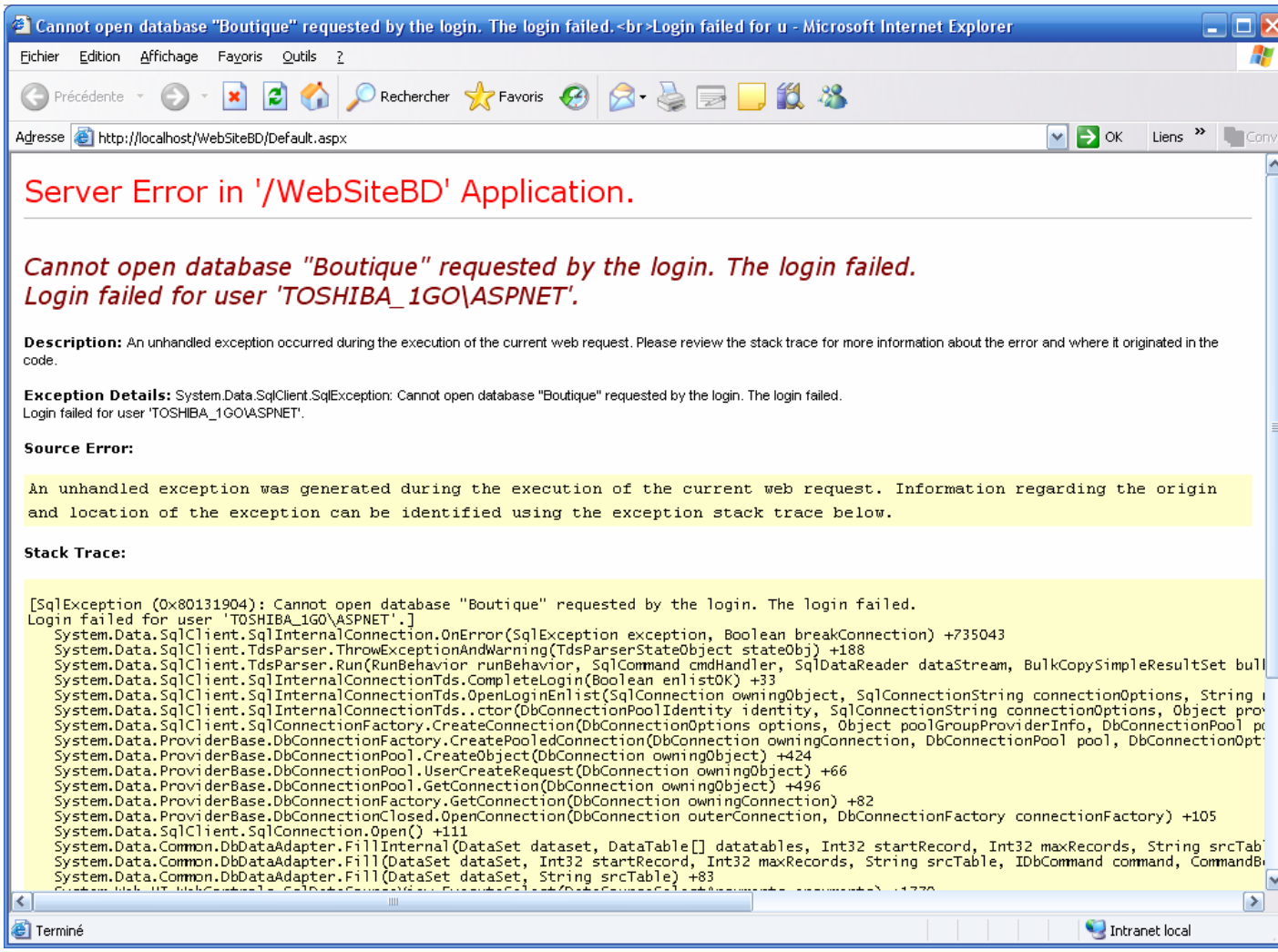
	idArticle	genreArticle	quantité	prix
Edit Delete Select	0	abc	0	0
Edit Delete Select	1	abc	1	0,1
Edit Delete Select	2	abc	2	0,2
Edit Delete Select	3	abc	3	0,3
Edit Delete Select	4	abc	4	0,4

SqlDataSource - SqlDataSource1

```
        <asp:GridView ID="GridViewBD" runat="server"
AutoGenerateColumns="False"
            BackColor="LightGoldenrodYellow"
            BorderColor="Tan" BorderWidth="1px" CellPadding="2"
            DataSourceID="SqlDataSource1" ForeColor="Black" GridLines="None"
            Width="402px" RowHeaderColumn="idArticle" ShowFooter="True"
            AutoGenerateDeleteButton="True"
            AutoGenerateEditButton="True"
            AutoGenerateSelectButton="True">
            . . .

        </asp:GridView>
```

Si le serveur SQL n'a pas les permissions adéquates pour ASP.NET

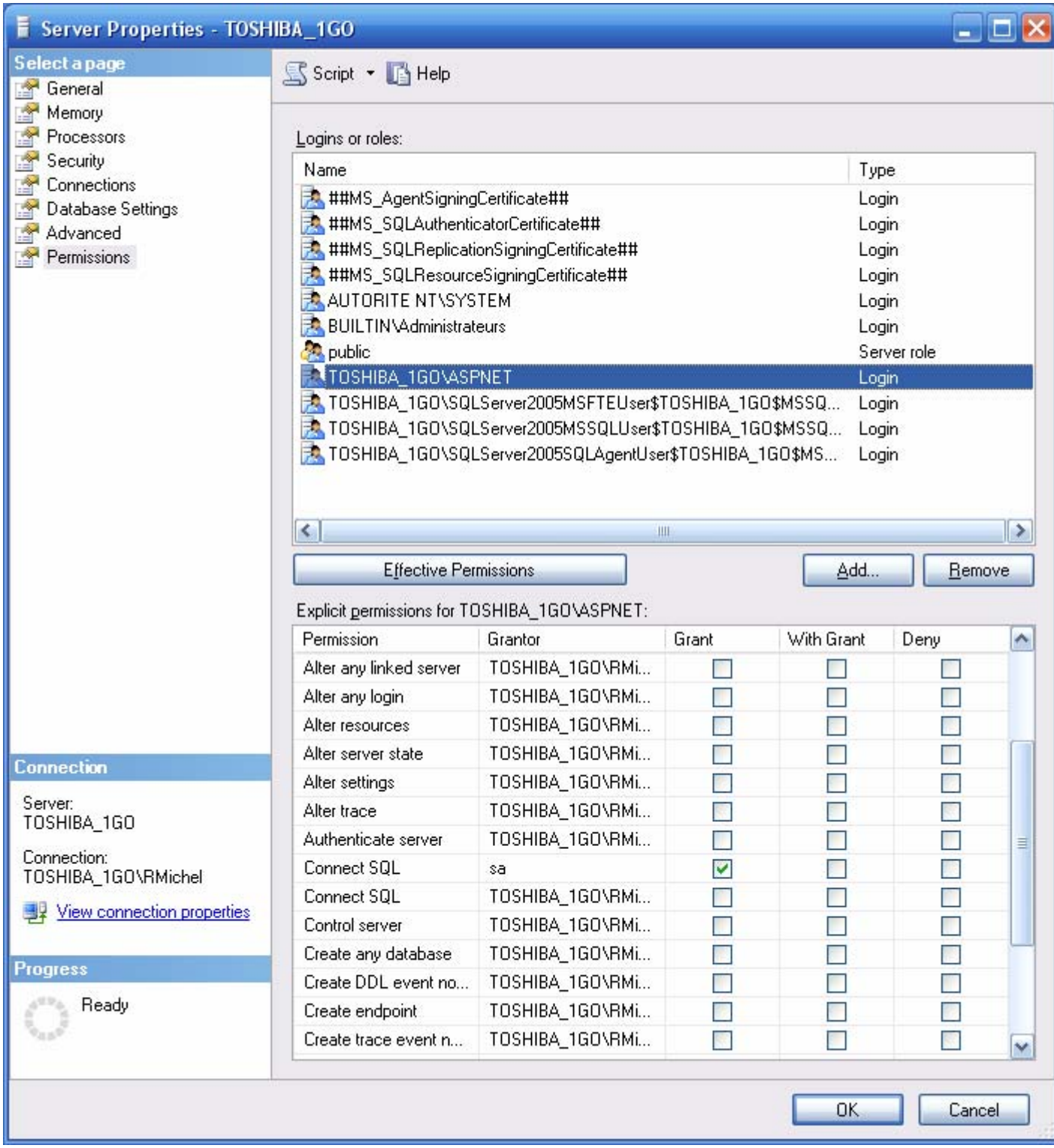
Voici le genre de message envoyé dans le navigateur (ici le serveur SQL se nomme **TOSHIBA_1GO**) :



Par défaut nous allons donner au serveur AspNet (dénommé : TOSHIBA_1GO\ASPNET) toutes les permissions pour pouvoir utiliser la BD avec Asp.Net :

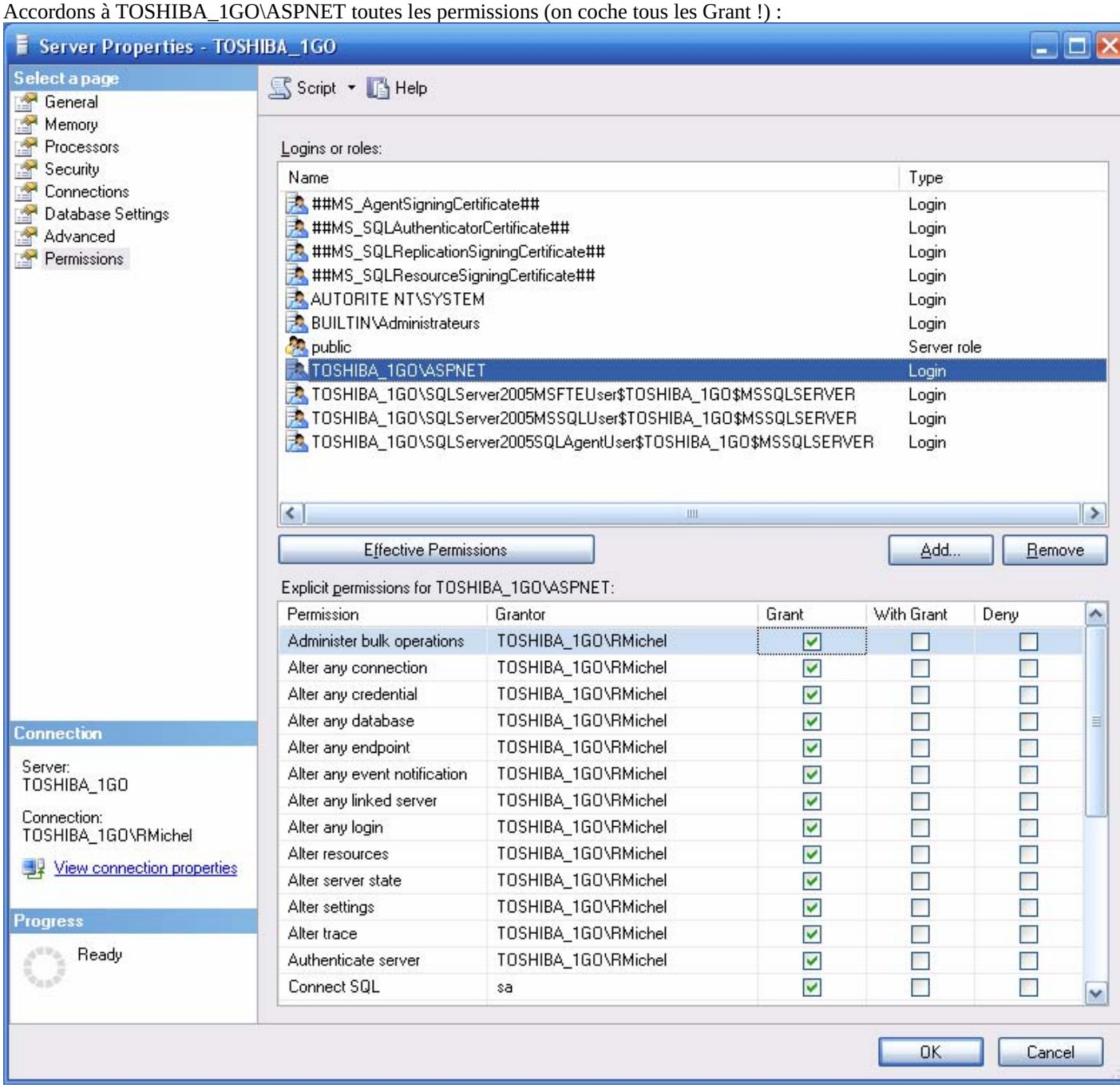
1°) sélectionner propriétés de la BD (ici Boutique.mdf)
2°) sélectionner page Permissions, puis le lien à cliquer est : View server permissions.

Les informations de permissions des serveurs sont affichées :



On note que par défaut, le serveur ASPNEt ici nommé TOSHIBA_1GO\ASPNET n'a pour l'instant qu'une seule permission accordée Connect SQL (Grant).

Accordons à TOSHIBA_1GO\ASPNET toutes les permissions (on coche tous les Grant !) :



En relançant le programme Asp.Net, le navigateur affiche maintenant :

Boutique

Test de GridView et BD Sql Server 2005 avec ASP.net 2.0

	idArticle	genreArticle	quantité	prix
Edit Delete Select	100	cahier	106	1,2000
Edit Delete Select	101	fourchette	57	0,9000
Edit Delete Select	102	cahier	120	2,0000
Edit Delete Select	103	serviette	28	1,4000
Edit Delete Select	104	verre	42	8,0000
Edit Delete Select	105	assiette	108	2,0000
Edit Delete Select	106	chemise	420	8,5000
Edit Delete Select	107	couteau	83	1,1000
Edit Delete Select	108	pantalon	27	45,0000
Edit Delete Select	**109**	**ski**	**25**	**12,5000**
Edit Delete Select	110	craie	1249	0,0400
Edit Delete Select	111	xcxxc	1000	100,0000

Rendre Delete actif

ligne sélectionnée = 9

Select

…

Autre vue de la même BD dans la page web :

	idArticle	genreArticle	quantité	prix
Edit Select	100	cahier	106	1,2000
Edit Select	101	fourchette	57	0,9000
Edit Select	102	cahier	120	2,0000
Edit Select	103	serviette	28	1,4000
Edit Select	104	verre	42	8,0000
Edit Select	105	assiette	108	2,0000

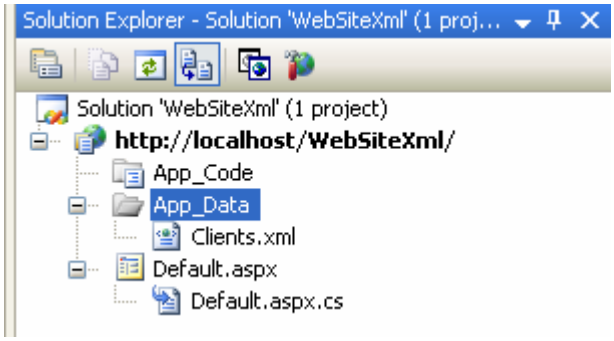
Des données hiérarchiques au format XML avec un XmlDataSource

Créer (ou utiliser) un fichier xml au format ci-dessous :

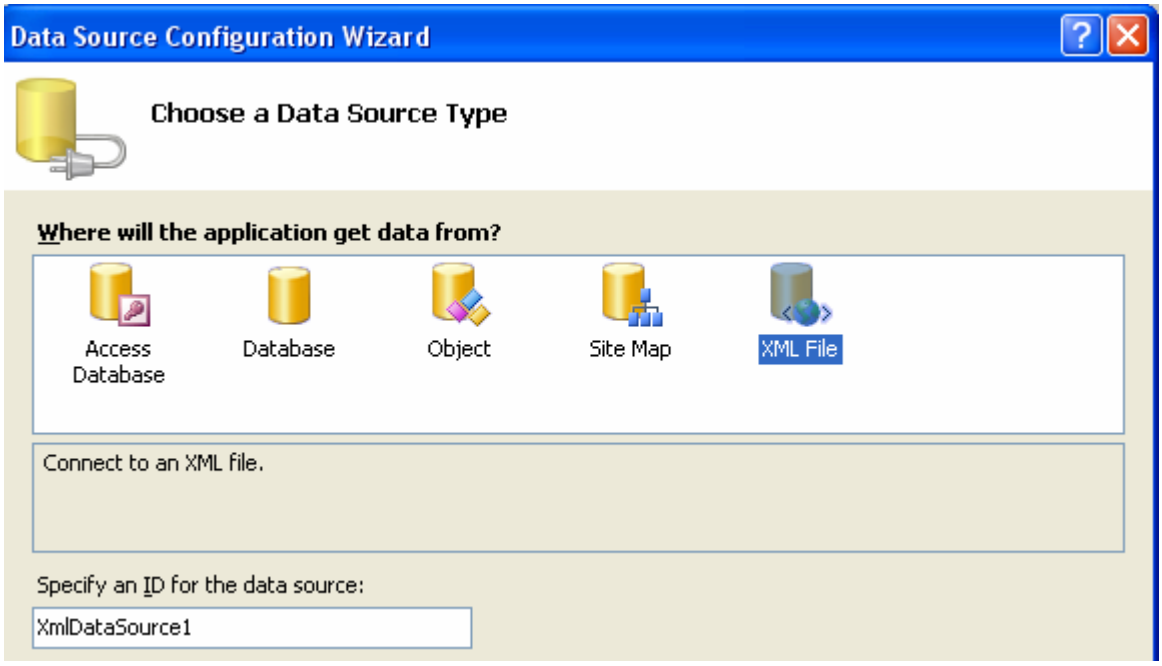
Start Page | App_Data/Clients.xml | Default.aspx.cs* | Default.aspx

```
<?xml version="1.0" encoding="utf-8" ?>
  <tableClient>
    <tclient idClient="1001" Nom="duval" Article="brosse" Montant="60.5"/>
    <tclient idClient="1002" Nom="dupont" Article="savon" Montant="140"/>
    <tclient idClient="1003" Nom="durand" Article="gant" Montant="24.8"/>
    <tclient idClient="1004" Nom="duchenaud" Article="serviette" Montant="35.7"/>
    <tclient idClient="1005" Nom="dumartin" Article="tapis" Montant="46"/>
  </tableClient>
```

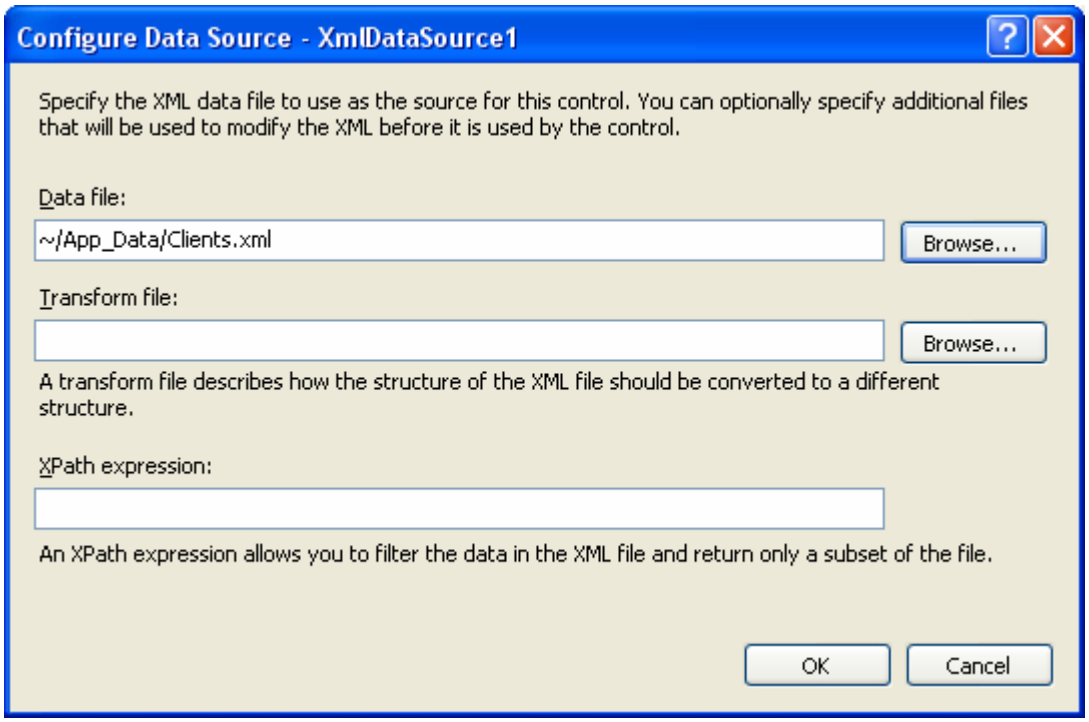
Ajouter ce fichier nommé par exemple "Clients.xml", au dossier **App_Data** :



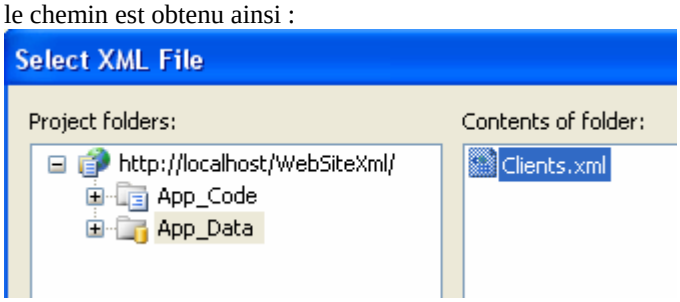
Ajouter un composant XmlDataSource à partir de la ToolBox :



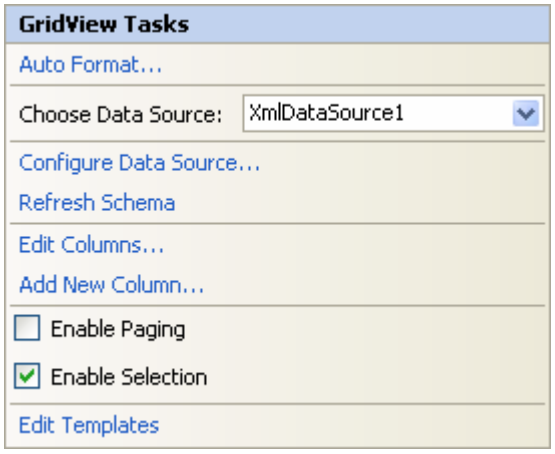
Configurer le XmlDataSource à partir du fichier Clients.xml :



le chemin est obtenu ainsi :



Il faut lier le GridView au XmlDataSource :



On obtient dès lors en mode construction l'affichage tabulaire du fichier Clients.xml :

Start Page | App_Data/Clients.xml | Default.aspx.cs* | **Default.aspx**

Le GridView sert à visualiser une table de données au format XML

	idClient	Nom	Article	Montant
Select	1001	duval	brosse	60.5
Select	1002	dupont	savon	140
Select	1003	durand	gant	24.8
Select	1004	duchenaud	serviette	35.7
Select	1005	dumartin	tapis	46

En exécutant le programme voici ce qui apparaît dans le navigateur utilisateur :

Untitled Page - Microsoft Internet Explorer

Fichier Edition Affichage Favoris Outils ?

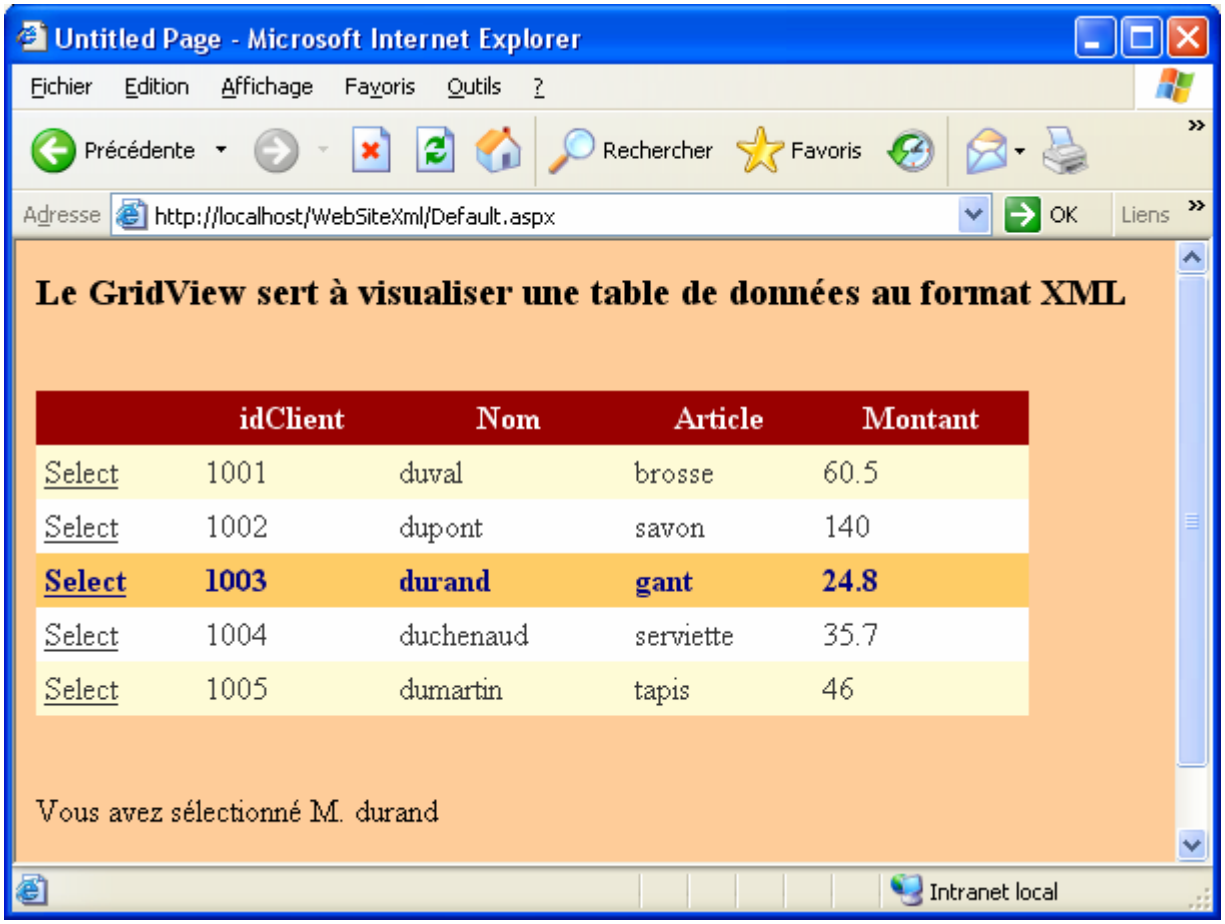
Précédente Rechercher Favoris

Adresse http://localhost/WebSiteXml/Default.aspx OK Liens

Le GridView sert à visualiser une table de données au format XML

	idClient	Nom	Article	Montant
Select	1001	duval	brosse	60.5
Select	1002	dupont	savon	140
Select	1003	durand	gant	24.8
Select	1004	duchenaud	serviette	35.7
Select	1005	dumartin	tapis	46

Terminé Intranet local



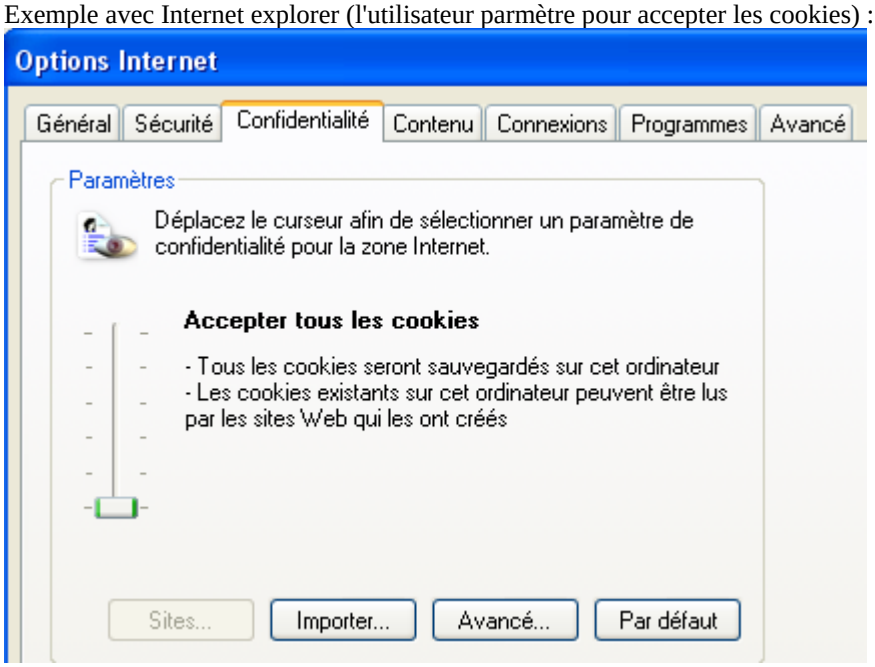

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    private int SelectedIndex;

    protected void Page_Load(object sender, EventArgs e)
    {
        if (IsPostBack)
        {
            SelectedIndex = (int)Page.Session["SelectedIndex"];
        }
        else
        {
            Page.Session["SelectedIndex"] = -1;
        }
    }
    protected void GridView1_SelectedIndexChanged(object sender, EventArgs e)
    {
        Page.Session["SelectedIndex"] = GridView1.SelectedIndex;
        LabelNom.Text = GridView1.SelectedRow.Cells[2].Text;
    }
}
```

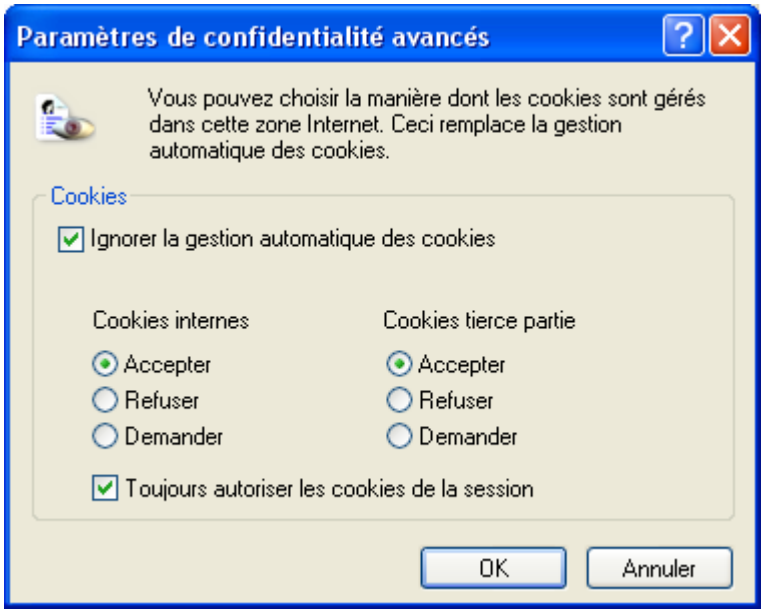
ASP.NET et Cookie

Les principes de sécurité d'Internet interdisent à tout programme d'accéder directement aux disques du client pour y stocker des fichiers ou bien même les lire, les seuls fichiers autorisésse dénomment des cookies.

Un **cookie** est obligatoirement un fichier texte ASCII dont la taille ne dépasse pas les 4 Ko et à durée de vie limitée, il est le seul type de fichier qu'une application web peut écrire et lire sur le disque dur de l'utilisateur , si l'utilisateur le permet par paramétrage de son navigateur.

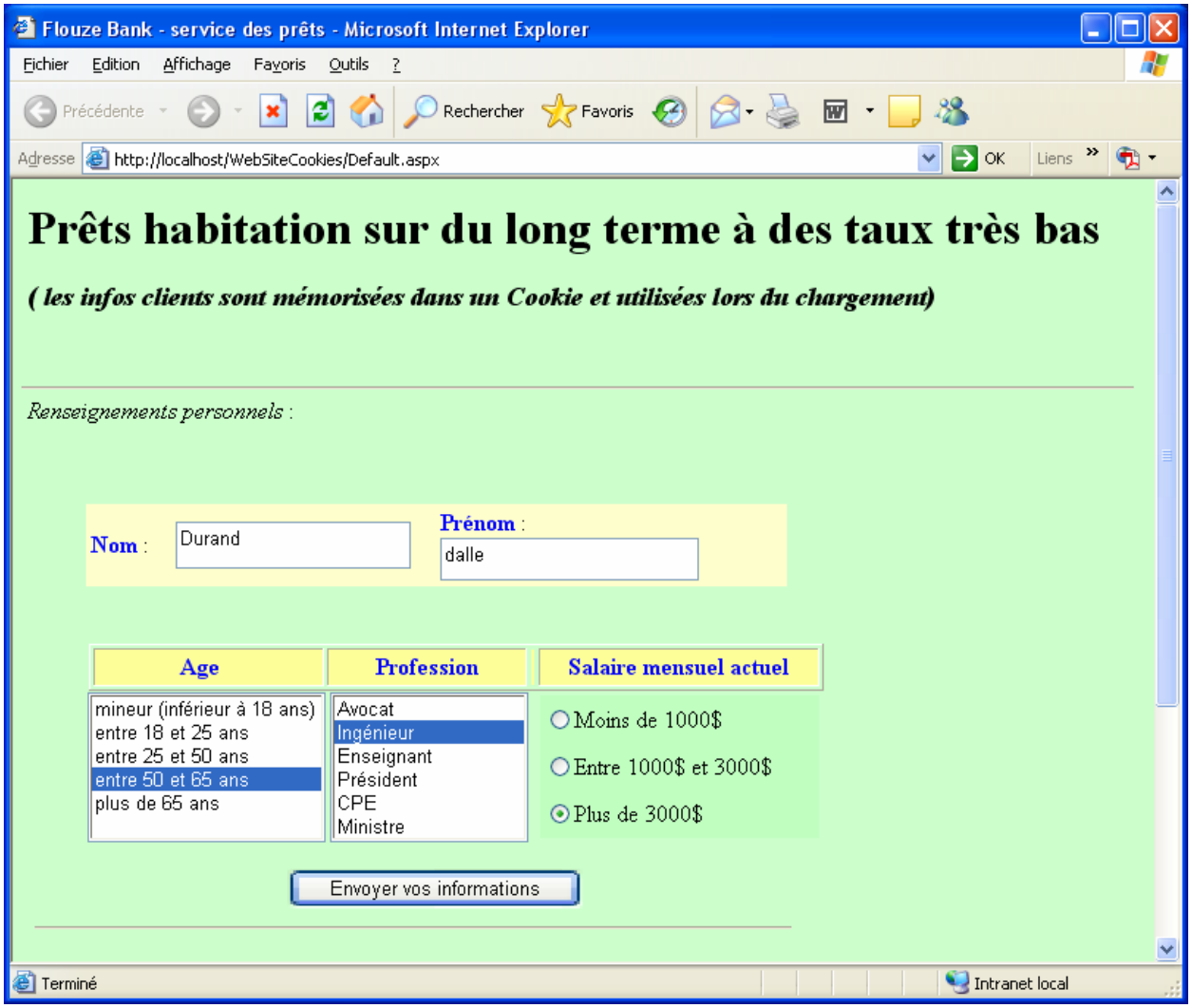
Exemple avec Internet explorer (l'utilisateur parmètre pour accepter les cookies) :



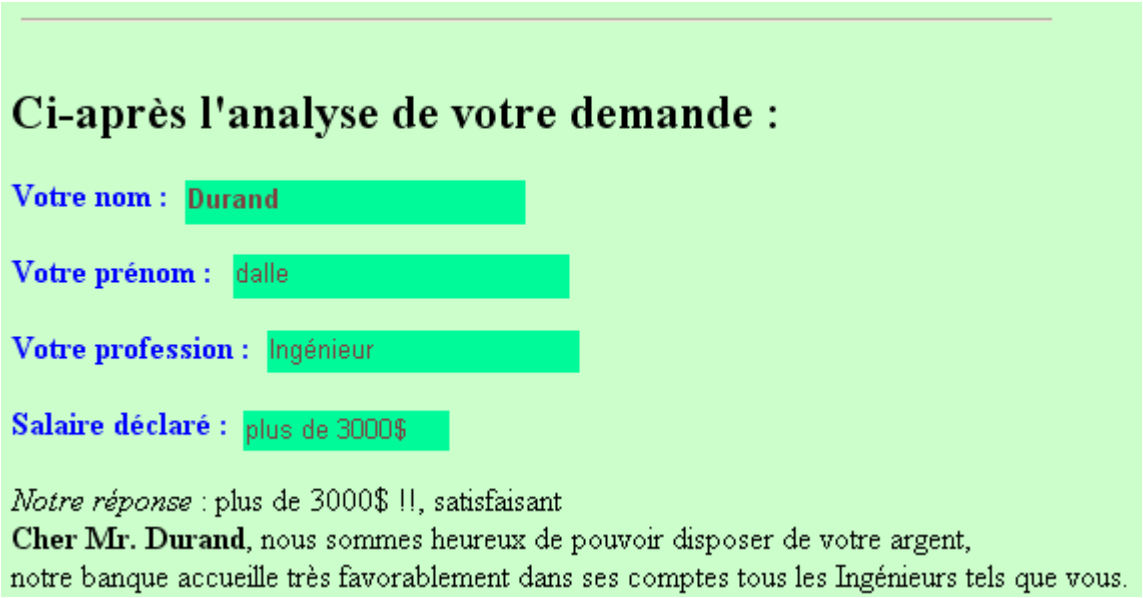


Dans cette éventualité, nous montrons une page web de compte bancaire qui récolte les données entrées (nom, prénom, caractéristiques du client) et qui les stocke dans un "cookie".

Soit une connexion à notre site web d'ouverture d'un compte en vue d'un prêt pour habitation de Mr Durand dalle âgé de 51 ans ingénieur de profession et pourvu d'un salaire mensuel de 5000 euros, le site web lui envoie sous forme d'une page html le formulaire de saisie ci-dessous :

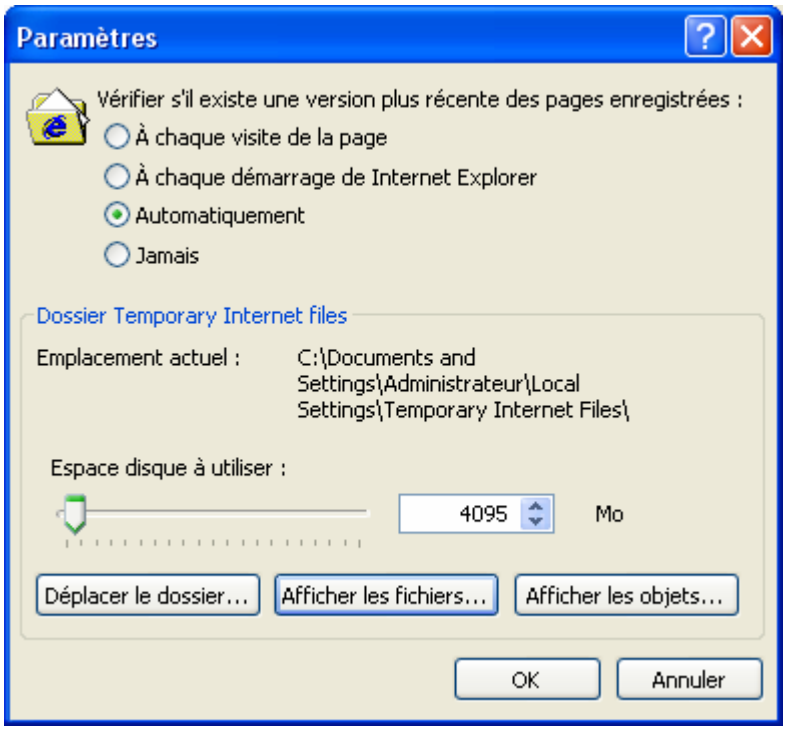


Après avoir saisie les renseignements demandés, Mr Durand dalle clique sur le bouton "envoyer vos informations" et le site web de la banque lui renvoie une nouvelle page html avec la réponse à sa demande :



A cet instant le site web a écrit et déposé sur le disque dur de Mr Durand un fichier texte (cookie) contenant les informations sur les 4 champs précédents (nom, prénom,…).

Par défaut les cookies sont automatiquement stockés dans le dossier "Temporary Internet Files" de la machine utilisateur sous son profil (dans l'exemple ci-dessous l'utilsateur se nomme "Administrateur") :



Localisation du cookie sur la machine

C:\Documents and Settings\Administrateur\Local Settings\Temporary Internet Files :

Nom	Adresse Internet	Type	Taille	Date d'expiration	Dernière modificatio.
Cookie:administrateur@localhost/	Cookie:administrateur@localhost/	Document texte	1 Ko	29/08/2006 08:25	28/08/2006 08:25
Default.aspx	http://localhost/WebSiteCookies...	HTML Document	6 Ko	Aucun	Aucun
ShowMe.bmp	ms-its:C:\PROGRA~1\ARCHIT~...	Image bitmap	2 Ko	Aucun	Aucun
copycode.gif	ms-itss:C:\Program Files\MSDN\...	Image GIF	1 Ko	Aucun	Aucun
footer.gif	ms-itss:C:\Program Files\MSDN\...	Image GIF	1 Ko	Aucun	Aucun
copycodeHighlight.gif	ms-itss:C:\Program Files\MSDN\...	Image GIF	1 Ko	Aucun	Aucun
drpdown.gif	ms-itss:C:\Program Files\MSDN\...	Image GIF	1 Ko	Aucun	Aucun
drpdown_orange.gif	ms-itss:C:\Program Files\MSDN\...	Image GIF	1 Ko	Aucun	Aucun
script.js	ms-itss:C:\Program Files\MSDN\...	JScript Script ...	25 Ko	Aucun	Aucun

Dans l'exemple, le fichier a été déposé le 28.08.2006 à 8h 25 sur la machine client, sa durée de vie est limitée à un jour, il expire le 29.08.2006 à 8h 25, ce qui veut dire qu'il sera effacé par le système à cette heure là le 29.08.2006.

Un double permanent (sauf si l'utilisateur décide de l'effacer) du cookie, se trouve dans le dossier Cookies sous le profil de l'utilisateur :

Contenu du fichier cookie sur la machine

identite
nom=Durand&prenom=dalle&profession=Ingénieur&salaire=plus de 3000$
localhost/

1024
4099591552
29805363
3395668048
29805162
*

Mr Durand coupe sa connexion au site de la banque et l'abandonne le cookie rste présent sur son disque. Si Mr Durand revient plus tard à 8h 38 sur le site de cette même banque, alors le programme web de la banque peut aller consulter le cookie présent sur son disque dur et récupérer les informations déjà entrées afin par exemple de les afficher dans la page HTML du navigateur de Mr Durand pour une éventuelle modification :

Dans C:\Documents and Settings\Administrateur\Local Settings\Temporary Internet Files :

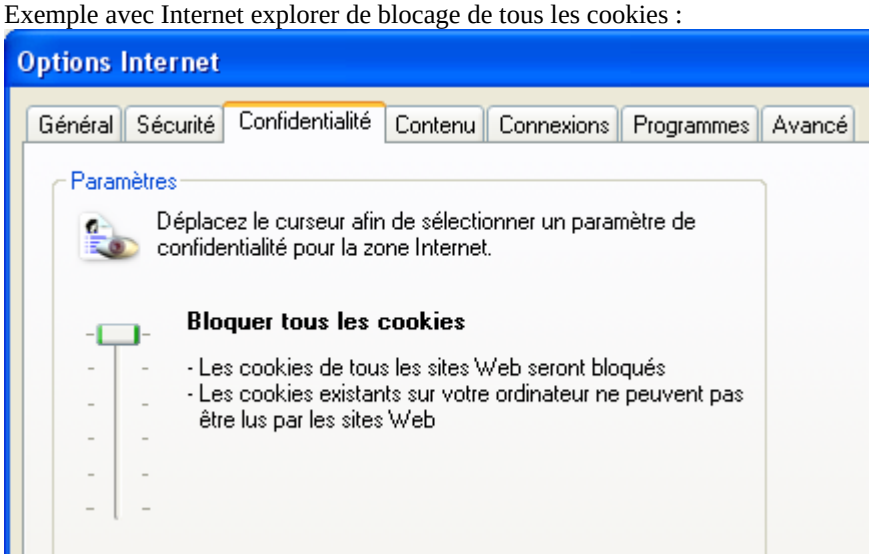
Nom	Date d'expiration	Dernière modificatio...	Dernier accès le
Default.aspx	Aucun	Aucun	28/08/2006 08:38
Cookie:administrateur@localhost/	29/08/2006 08:25	28/08/2006 08:25	28/08/2006 08:38
ShowMe.bmp	Aucun	Aucun	18/08/2006 20:36
copycode.gif	Aucun	Aucun	08/08/2006 08:57
footer.gif	Aucun	Aucun	08/08/2006 08:57
copycodeHighlight.gif	Aucun	Aucun	08/08/2006 08:57
drpdown.gif	Aucun	Aucun	08/08/2006 08:57
drpdown_orange.gif	Aucun	Aucun	08/08/2006 08:57
script.js	Aucun	Aucun	08/08/2006 08:57

Le système a bien marqué le cookie comme ayant été accédé la dernière fois à 8h 38.

Attention

Une application web ne doit pas être dépendante des cookies, car l'utilisateur peut refuser les cookies et l'application risque de ne pas fonctionner correctement :

Exemple avec Internet explorer de blocage de tous les cookies :



Comment programmer et utiliser le cookie précédent

Code Asp.Net de la page Default.aspx construite :

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Flouze Bank - service des prêts</title>
</head>
<body bgcolor="#ccffcc">
   <form id="form1" runat="server">
   <div>
   <h1>
<asp:listbox id="Select2" style="Z-INDEX: 105; LEFT: 51px; POSITION: absolute; TOP: 349px"
runat="server"Width="162px" Height="102px">
<asp:ListItem Value="0">mineur (inf&#233;rieur &#224; 18 ans)</asp:ListItem>
<asp:ListItem Value="1" Selected="True">entre 18 et 25 ans</asp:ListItem>
<asp:ListItem Value="2">entre 25 et 50 ans</asp:ListItem>
<asp:ListItem Value="3">entre 50 et 65 ans</asp:ListItem>
<asp:ListItem Value="4">plus de 65 ans</asp:ListItem>
</asp:listbox>
<asp:textbox id="TextSalaire" style="Z-INDEX: 112; LEFT: 126px; POSITION: absolute; TOP: 704px"
          runat="server" Width="101px" Height="18px" BackColor="MediumSpringGreen" ReadOnly="True"
BorderWidth="0px" ForeColor="#804040">
</asp:textbox>
<asp:textbox id="TextProfession" style="Z-INDEX: 111; LEFT: 138px; POSITION: absolute; TOP: 664px"
runat="server" Width="154px" Height="19px" BackColor="MediumSpringGreen" ReadOnly="True"
BorderWidth="0px" ForeColor="#804040">
</asp:textbox>
```

```
<asp:textbox id="TextPrenom" style="Z-INDEX: 110; LEFT: 121px; POSITION: absolute; TOP: 626px"
runat="server" Width="166px" Height="20px" BackColor="MediumSpringGreen" ReadOnly="True"
BorderWidth="0px" ForeColor="#804040">
</asp:textbox>
<asp:textbox id="TextNom" style="Z-INDEX: 109; LEFT: 97px; POSITION: absolute; TOP: 589px"
runat="server" Width="168px" Height="20px" BackColor="MediumSpringGreen" ReadOnly="True"
BorderWidth="0px" ForeColor="#804040" Font-Bold="True">
</asp:textbox>
<asp:listbox id="Select1" style="Z-INDEX: 106; LEFT: 216px; POSITION: absolute; TOP: 349px" runat="server"
Width="135px" Height="105px">
<asp:ListItem Value="0">Avocat</asp:ListItem>
<asp:ListItem Value="1">Ing&#233;nieur</asp:ListItem>
<asp:ListItem Value="2" Selected="True">Enseignant</asp:ListItem>
<asp:ListItem Value="3">Pr&#233;sident</asp:ListItem>
<asp:ListItem Value="4">CPE</asp:ListItem>
<asp:ListItem Value="5">Ministre</asp:ListItem>
</asp:listbox>
Prêts habitation sur du long terme à des taux très bas</h1>
<h3><i>( les infos clients sont mémorisées dans un Cookie et utilisées lors du  chargement)</i></h3>
<HR style="Z-INDEX: 100; LEFT: 6px; POSITION: absolute; TOP: 141px">
<TABLE id="Table1" style="Z-INDEX: 101; LEFT: 50px; WIDTH: 477px; POSITION: absolute; TOP: 221px;
HEIGHT: 34px" bgcolor="#ffffcc">
<TR>
<TD bgColor="#ffffcc" style="width: 42px"><B><FONT color="blue">Nom</FONT></B> :</TD>
<TD bgColor="#ffffcc">  
 <asp:textbox id="Text1" runat="server" Width="154px" Height="26px">
</asp:textbox>    
</TD>
<TD bgColor="#ffffcc"><B><FONT color="blue">Prénom</FONT></B> :
<asp:textbox id="Text2" runat="server" Width="170px" Height="23px">
</asp:textbox>
</TD>
</TR>
</TABLE>
<br>
<div align="left"><br>
<i>Renseignements personnels</i> :
<br><br /><br /><br>
</div>
<asp:button id="Button1" style="Z-INDEX: 102; LEFT: 187px; POSITION: absolute; TOP: 469px" runat="server"
Width="201px" Height="27px" Text="Envoyer vos informations" OnClick="Button1_Click">
</asp:button>
<TABLE id="Table2" style="Z-INDEX: 104; LEFT: 52px; WIDTH: 500px; POSITION: absolute; TOP: 316px;
HEIGHT: 32px"cols="1" border="1">
<TR>
<TH style="WIDTH: 153px" bgColor="#ffff99">
<FONT color="blue">Age   </FONT>
</TH>
<TH style="WIDTH: 138px" bgColor="#ffff99">
<FONT color="blue">Profession</FONT>
</TH>
<TH bgColor="#ffff99">
</TH>
<TH style="WIDTH: 200px" bgColor="#ffff99">
<FONT color="blue">Salaire mensuel actuel</FONT>
</TH>
</TR>
</TABLE>
<asp:radiobuttonlist id="RadioButtonList1" style="Z-INDEX: 107; LEFT: 359px; POSITION: absolute; TOP:
351px" runat="server" Width="190px" Height="97px" BackColor="#C0FFC0">
```

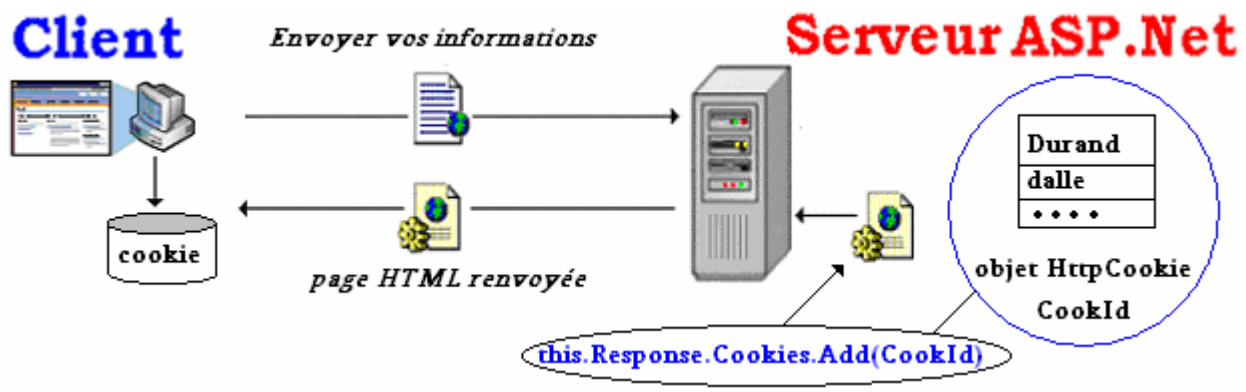
```
<asp:ListItem Value="0">Moins de 1000$</asp:ListItem>
<asp:ListItem Value="1" Selected="True">Entre 1000$ et 3000$</asp:ListItem>
<asp:ListItem Value="2">Plus de 3000$</asp:ListItem>
</asp:radiobuttonlist>
<HR style="Z-INDEX: 108; LEFT: 15px; WIDTH: 68%; POSITION: absolute; TOP: 508px; HEIGHT: 2px"
SIZE="2">
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
                                <br>
<h2 id="H11">  </h2>
 
<h2> Ci-après l'analyse de votre demande :</h2>
                                <b><font color="blue">Votre nom :</font></b>
                                <br>
                                <br>
                                <b><font color="blue">Votre prénom :</font> </b>
                                <br>
                                <br>
                                <b><font color="blue">Votre profession : </font></b>
                                <br>
                                <br>
                                <b><font color="blue">Salaire déclaré :</font></b>
                                <br>
                                <br>
                                <i>Notre réponse</i> :
                                <%= texteRep %>
   </div>
   </form>
</body>
</html>
```

C'est un objet de la classe `HttpCookie` que nous nommons CookId, qui prend en charge dans .Net la gestion des cookies, syntaxiquement un tel objet se programme comme un objet Session déjà vu à travers un adressage indirect.

Lorsque l'utilisateur Mr Durand dalle clique sur le bouton "envoyer vos informations", le serveur exécute le code C# ci-après qui permet de créer le fichier de cookie sur la machine de Mr Durand :

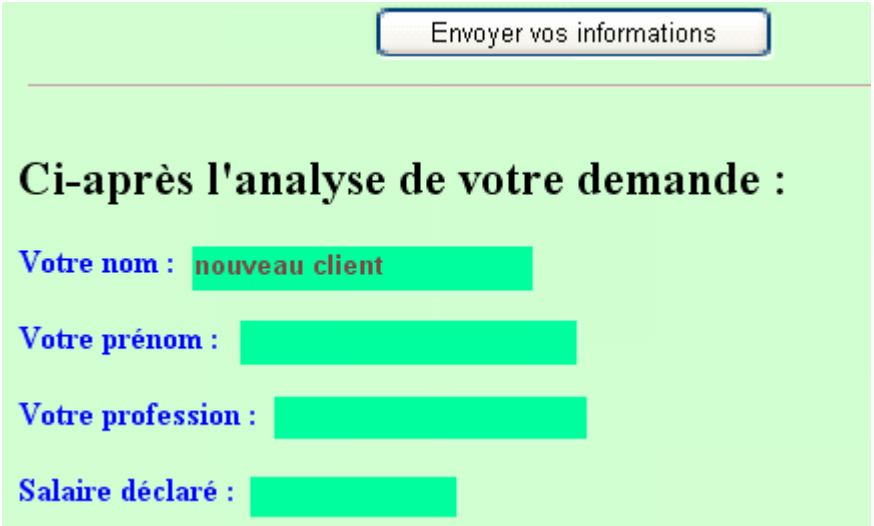
```
//-- Persistance des données : (un seul cookie pour un site et pas plus de 4Ko de données)
    string Reponse = "";
    …..
    HttpCookie CookId = new HttpCookie("identite");
    CookId["nom"] = VotreNom; // création du champ "nom" dans le cookie avec sa donnée
    CookId["prenom"] = VotrePreNom; // création du champ "prenom" dans le cookie avec sa donnée
    CookId["profession"] = votreprofession; // création du champ "profession" dans le cookie avec sa donnée
    CookId["salaire"] = salaire; // création du champ "salaire" dans le cookie avec sa donnée
    this.Session["Reponse"] = Reponse;
    CookId.Expires = DateTime.Now.AddDays(1); // date d'expiration réglée sur jour actuel +1

    //-- validation et stockage :
    bool isChampValid = VotreNom.Length * VotrePreNom.Length != 0;
```

```
if (isChampValid)//si tous les champs sont remplis
   this.Response.Cookies.Add(CookId);// on envoie la réponse http au client : ajoutez le cookie
```



Lors du chargement de la page sur le serveur, le programme teste la présence du cookie sur la machine web, s'il n'y a pas de cookie présentement associé à l'application, le serveur ne renseigne que le champ TextNom avec le texte "nouveau client" :

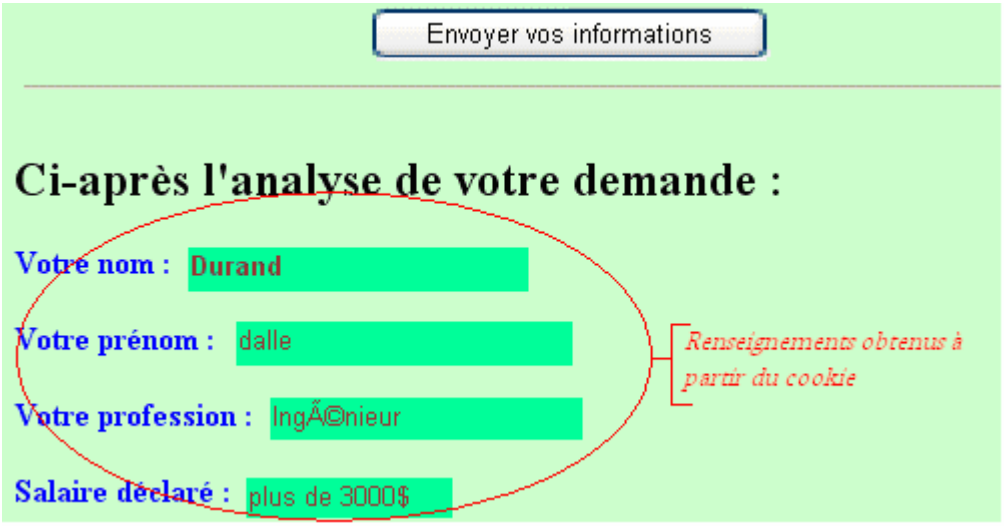
Affichage dans le navigateur client :



Si lors du chargement de la page sur le serveur, le programme teste la présence du cookie sur la machine web et que celui-ci existe, le serveur renseigne tous les champs TextNom, TextPrenom, TextProfession, TextSalaire à partir des données lues dans le cookie :

```
protected void Page_Load(object sender, System.EventArgs e)
 {
    HttpCookie CookId = this.Request.Cookies["identite"];
    if (CookId != null)
    {
       TextNom.Text = this.Request.Cookies["identite"]["nom"];
       TextPrenom.Text = this.Request.Cookies["identite"]["prenom"];
       TextProfession.Text = this.Request.Cookies["identite"]["profession"];
       TextSalaire.Text = this.Request.Cookies["identite"]["salaire"];
       texteRep = Convert.ToString(this.Session["Reponse"]);
    }
    else TextNom.Text = "nouveau client";
 }
```

Affichage dans le navigateur client :



Code C# de la page Default.aspx.cs associée :

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
   private String[ ] tprofession = { "Avocat", "Ingénieur", "Enseignant", "Président", "CPE", "Ministre" };
   private String[ ] tage = { "age < 18", "18 < age < 25", "25 < age < 50", "50 < age < 65", "age > 65" };
   protected string texteRep;

   protected void Page_Load(object sender, System.EventArgs e)
   {
      HttpCookie CookId = this.Request.Cookies["identite"];
      if (CookId != null)
      {
         TextNom.Text = this.Request.Cookies["identite"]["nom"];
         TextPrenom.Text = this.Request.Cookies["identite"]["prenom"];
         TextProfession.Text = this.Request.Cookies["identite"]["profession"];
         TextSalaire.Text = this.Request.Cookies["identite"]["salaire"];
         texteRep = Convert.ToString(this.Session["Reponse"]);
      }
      else TextNom.Text = "nouveau client";
   }
   protected void Button1_Click(object sender, EventArgs e)
   {
      string VotreNom = "";
      string VotrePreNom = "";
      string votreprofession = "";
```

```
        string votreAge = "";
        string salaire = "";
        string Reponse = "";
        if (Text1.Text != null && Text1.Text.Length != 0)
        {
            VotreNom = Text1.Text;
            if (Text2.Text != null && Text2.Text.Length != 0)
            {
                VotrePreNom = Text2.Text;
                votreAge = tage[Select2.SelectedIndex];
                votreprofession = tprofession[Select1.SelectedIndex];

                switch (RadioButtonList1.SelectedIndex)
                {
                    case 0: Reponse = "<br>moins de 1000$, insuffisant !! Dossier rejeté !";
                        salaire = "moins de 1000$"; break;
                    case 1: Reponse = "<br>moins de 3000$ ?, à peine suffisant, nous conservons votre dossier pour étude
approfondie.";
                        salaire = "moins de 3000$"; break;
                    case 2: Reponse = "plus de 3000$ !!, satisfaisant";
                        if (votreprofession != "CPE")
                            Reponse += "<br><b>Cher Mr. " + VotreNom + "</b>, nous sommes heureux de pouvoir disposer
de votre argent,<br>        notre banque accueille très favorablement dans ses comptes tous les " + votreprofession +
"s tels que vous.";
                        salaire = "plus de 3000$";
                        break;
                }
                if (votreprofession == "CPE")
                    if (RadioButtonList1.SelectedIndex == 2)
                        Reponse += "<br><b>Cher Mr. " + VotreNom + "</b>, nous sommes heureux de pouvoir disposer
de votre argent,<br>        notre banque accueille malgré tout dans ses comptes les " + votreprofession + " tels que
vous."
                      + " <br>Toutefois votre situation précaire ne nous permet " +
                    "d'envisager un prêt a l'habitat que sur 3 mois maximum et à un taux majoré de 200% pour risque de
solvabilité." +
                    "<br>Avec nos conditions vous pourrez ainsi acheter un magnifique carton renforçé, dans lequel il vous
sera possible de faire la manche pour nous rembourser.";
                    else
                        if (RadioButtonList1.SelectedIndex == 1)
                            Reponse += " <br>Toutefois vue votre situation précaire votre dossier vient d'être
malencontreusement jeté dans notre poubelle." +
                            "<br>Revenez nous voir avec un CDI.";
                        else
                            if (RadioButtonList1.SelectedIndex == 0)
                                Reponse += " <br>Avec votre situation précaire, vous osez en plus vous payer un logement !";
            }
            else
            {
                Reponse = "<br><i><font color=#ff0066>Votre prénom est incorrectement entré !</font></i>";
            }
        }
        else
        {
            Reponse = "<br><i><font color=#ff0066>Vous n'avez pas saisi correctement votre nom !!</font></i>";
        }
        TextNom.Text = VotreNom;
        TextPrenom.Text = VotrePreNom;
        TextProfession.Text = votreprofession;
        TextSalaire.Text = salaire;
        texteRep = Reponse;
```

```
        //-- Persistance des données :
        // (un seul cookie pour un site et pas plus de 4Ko de données)
        HttpCookie CookId = new HttpCookie("identite");
        CookId["nom"] = VotreNom;
        CookId["prenom"] = VotrePreNom;
        CookId["profession"] = votreprofession;
        CookId["salaire"] = salaire;
        this.Session["Reponse"] = Reponse;
        CookId.Expires = DateTime.Now.AddDays(1);

        //-- validation et stockage :
        bool isChampValid = VotreNom.Length * VotrePreNom.Length != 0;
        if (isChampValid)//si tous les champs sont remplis
            this.Response.Cookies.Add(CookId);// on envoie la réponse http au client
    }
}
```

ANNEXE javascript

Sources des fichiers WebResource.axd?d=….. auto générés et se trouvaent dans "C:\Documents and Settings\Administrateur\Local Settings\Temporary Internet Files" :

Famille de fonctions javascript de gestion de menus auto-générée

```
var __rootMenuItem;
var __menuInterval;
var __scrollPanel;
var __disappearAfter = 500;
function Menu_ClearInterval() {
   if (__menuInterval) {
      window.clearInterval(__menuInterval);
   }
}
function Menu_Collapse(item) {
   Menu_SetRoot(item);
   if (__rootMenuItem) {
      Menu_ClearInterval();
      if (__disappearAfter >= 0) {
         __menuInterval = window.setInterval("Menu_HideItems()", __disappearAfter);
      }
   }
}
function Menu_Expand(item, horizontalOffset, verticalOffset, hideScrollers) {
   Menu_ClearInterval();
   var tr = item.parentNode.parentNode.parentNode.parentNode.parentNode;
   var horizontal = true;
   if (!tr.id) {
      horizontal = false;
      tr = tr.parentNode;
   }
   var child = Menu_FindSubMenu(item);
   if (child) {
      var data = Menu_GetData(item);
      if (!data) {
         return null;
      }
      child.rel = tr.id;
      child.x = horizontalOffset;
      child.y = verticalOffset;
      if (horizontal) child.pos = "bottom";
      PopOut_Show(child.id, hideScrollers, data);
   }
   Menu_SetRoot(item);
   if (child) {
      if (!document.body.__oldOnClick && document.body.onclick) {
         document.body.__oldOnClick = document.body.onclick;
      }
      if (__rootMenuItem) {
         document.body.onclick = Menu_HideItems;
      }
   }
   Menu_ResetSiblings(tr);
   return child;
}
```

```
function Menu_FindMenu(item) {
   if (item && item.menu) return item.menu;
   var tr = item.parentNode.parentNode.parentNode.parentNode.parentNode;
   if (!tr.id) {
      tr = tr.parentNode;
   }
   for (var i = tr.id.length - 1; i >= 0; i--) {
      if (tr.id.charAt(i) < '0' || tr.id.charAt(i) > '9') {
         var menu = WebForm_GetElementById(tr.id.substr(0, i));
         if (menu) {
            item.menu = menu;
            return menu;
         }
      }
   }
   return null;
}
function Menu_FindNext(item) {
   var a = WebForm_GetElementByTagName(item, "A");
   var parent = Menu_FindParentContainer(item);
   var first = null;
   if (parent) {
      var links = WebForm_GetElementsByTagName(parent, "A");
      var match = false;
      for (var i = 0; i < links.length; i++) {
         var link = links[i];
         if (Menu_IsSelectable(link)) {
            if (Menu_FindParentContainer(link) == parent) {
               if (match) {
                  return link;
               }
               else if (!first) {
                  first = link;
               }
            }
            if (!match && link == a) {
               match = true;
            }
         }
      }
   }
   return first;
}
function Menu_FindParentContainer(item) {
   if (item.menu_ParentContainerCache) return item.menu_ParentContainerCache;
   var a = (item.tagName.toLowerCase() == "a") ? item : WebForm_GetElementByTagName(item, "A");
   var menu = Menu_FindMenu(a);
   if (menu) {
      var parent = item;
      while (parent && parent.tagName &&
         parent.id != menu.id &&
         parent.tagName.toLowerCase() != "div") {
         parent = parent.parentNode;
      }
      item.menu_ParentContainerCache = parent;
      return parent;
   }
}
function Menu_FindParentItem(item) {
   var parentContainer = Menu_FindParentContainer(item);
```

```
    var parentContainerID = parentContainer.id;
    var len = parentContainerID.length;
    if (parentContainerID && parentContainerID.substr(len - 5) == "Items") {
        var parentItemID = parentContainerID.substr(0, len - 5);
        return WebForm_GetElementById(parentItemID);
    }
    return null;
}
function Menu_FindPrevious(item) {
    var a = WebForm_GetElementByTagName(item, "A");
    var parent = Menu_FindParentContainer(item);
    var last = null;
    if (parent) {
        var links = WebForm_GetElementsByTagName(parent, "A");
        for (var i = 0; i < links.length; i++) {
            var link = links[i];
            if (Menu_IsSelectable(link)) {
                if (link == a && last) {
                    return last;
                }
                if (Menu_FindParentContainer(link) == parent) {
                    last = link;
                }
            }
        }
    }
    return last;
}
function Menu_FindSubMenu(item) {
    var tr = item.parentNode.parentNode.parentNode.parentNode.parentNode;
    if (!tr.id) {
        tr=tr.parentNode;
    }
    return WebForm_GetElementById(tr.id + "Items");
}
function Menu_Focus(item) {
    if (item && item.focus) {
        var pos = WebForm_GetElementPosition(item);
        var parentContainer = Menu_FindParentContainer(item);
        if (!parentContainer.offset) {
            parentContainer.offset = 0;
        }
        var posParent = WebForm_GetElementPosition(parentContainer);
        var delta;
        if (pos.y + pos.height > posParent.y + parentContainer.offset + parentContainer.clippedHeight) {
            delta = pos.y + pos.height - posParent.y - parentContainer.offset - parentContainer.clippedHeight;
            PopOut_Scroll(parentContainer, delta);
        }
        else if (pos.y < posParent.y + parentContainer.offset) {
            delta = posParent.y + parentContainer.offset - pos.y;
            PopOut_Scroll(parentContainer, -delta);
        }
        PopOut_HideScrollers(parentContainer);
        item.focus();
    }
}
function Menu_GetData(item) {
    if (!item.data) {
        var a = (item.tagName.toLowerCase() == "a" ? item : WebForm_GetElementByTagName(item, "a"));
        var menu = Menu_FindMenu(a);
```

```
        try {
            item.data = eval(menu.id + "_Data");
        }
        catch(e) {}
    }
    return item.data;
}
function Menu_HideItems(items) {
    if (document.body.__oldOnClick) {
        document.body.onclick = document.body.__oldOnClick;
        document.body.__oldOnClick = null;
    }
    Menu_ClearInterval();
    if (!items || ((typeof(items.tagName) == "undefined") && (items instanceof Event))) {
        items = __rootMenuItem;
    }
    var table = items;
    if ((typeof(table) == "undefined") || (table == null) || !table.tagName || (table.tagName.toLowerCase() !=
"table")) {
        table = WebForm_GetElementByTagName(table, "TABLE");
    }
    if ((typeof(table) == "undefined") || (table == null) || !table.tagName || (table.tagName.toLowerCase() !=
"table")) {
        return;
    }
    var rows = table.rows ? table.rows : table.firstChild.rows;
    var isVertical = false;
    for (var r = 0; r < rows.length; r++) {
        if (rows[r].id) {
            isVertical = true;
            break;
        }
    }
    var i, child, nextLevel;
    if (isVertical) {
        for(i = 0; i < rows.length; i++) {
            if (rows[i].id) {
                child = WebForm_GetElementById(rows[i].id + "Items");
                if (child) {
                    Menu_HideItems(child);
                }
            }
            else if (rows[i].cells[0]) {
                nextLevel = WebForm_GetElementByTagName(rows[i].cells[0], "TABLE");
                if (nextLevel) {
                    Menu_HideItems(nextLevel);
                }
            }
        }
    }
    else {
        for(i = 0; i < rows[0].cells.length; i++) {
            if (rows[0].cells[i].id) {
                child = WebForm_GetElementById(rows[0].cells[i].id + "Items");
                if (child) {
                    Menu_HideItems(child);
                }
            }
            else {
                nextLevel = WebForm_GetElementByTagName(rows[0].cells[i], "TABLE");
```

```
            if (nextLevel) {
                Menu_HideItems(rows[0].cells[i].firstChild);
            }
        }
    }
  }
  if (items && items.id) {
    PopOut_Hide(items.id);
  }
}
function Menu_HoverDisabled(item) {
  var node = (item.tagName.toLowerCase() == "td") ?
    item:
    item.cells[0];
  var data = Menu_GetData(item);
  if (!data) return;
  node = WebForm_GetElementByTagName(node, "table").rows[0].cells[0].childNodes[0];
  if (data.disappearAfter >= 200) {
    __disappearAfter = data.disappearAfter;
  }
  Menu_Expand(node, data.horizontalOffset, data.verticalOffset);
}
function Menu_HoverDynamic(item) {
  var node = (item.tagName.toLowerCase() == "td") ?
    item:
    item.cells[0];
  var data = Menu_GetData(item);
  if (!data) return;
  var nodeTable = WebForm_GetElementByTagName(node, "table");
  if (data.hoverClass) {
    nodeTable.hoverClass = data.hoverClass;
    WebForm_AppendToClassName(nodeTable, data.hoverClass);
  }
  node = nodeTable.rows[0].cells[0].childNodes[0];
  if (data.hoverHyperLinkClass) {
    node.hoverHyperLinkClass = data.hoverHyperLinkClass;
    WebForm_AppendToClassName(node, data.hoverHyperLinkClass);
  }
  if (data.disappearAfter >= 200) {
    __disappearAfter = data.disappearAfter;
  }
  Menu_Expand(node, data.horizontalOffset, data.verticalOffset);
}
function Menu_HoverRoot(item) {
  var node = (item.tagName.toLowerCase() == "td") ?
    item:
    item.cells[0];
  var data = Menu_GetData(item);
  if (!data) {
    return null;
  }
  var nodeTable = WebForm_GetElementByTagName(node, "table");
  if (data.staticHoverClass) {
    nodeTable.hoverClass = data.staticHoverClass;
    WebForm_AppendToClassName(nodeTable, data.staticHoverClass);
  }
  node = nodeTable.rows[0].cells[0].childNodes[0];
  if (data.staticHoverHyperLinkClass) {
    node.hoverHyperLinkClass = data.staticHoverHyperLinkClass;
    WebForm_AppendToClassName(node, data.staticHoverHyperLinkClass);
```

```
    }
    return node;
}
function Menu_HoverStatic(item) {
    var node = Menu_HoverRoot(item);
    var data = Menu_GetData(item);
    if (!data) return;
    __disappearAfter = data.disappearAfter;
    Menu_Expand(node, data.horizontalOffset, data.verticalOffset);
}
function Menu_IsHorizontal(item) {
    if (item) {
        var
        a = ((item.tagName && (item.tagName.toLowerCase == "a")) ? item :
WebForm_GetElementByTagName(item, "A"));
        if (!a) {
            return false;
        }
        var td = a.parentNode.parentNode.parentNode.parentNode.parentNode;
        if (td.id) {
            return true;
        }
    }
    return false;
}
function Menu_IsSelectable(link) {
    return (link && link.href)
}
function Menu_Key(item) {
    var event;
    if (window.event) {
        event = window.event;
    }
    else {
        event = item;
        item = event.currentTarget;
    }
    var key = (event ? event.keyCode : -1);
    var data = Menu_GetData(item);
    if (!data) return;
    var horizontal = Menu_IsHorizontal(item);
    var a = WebForm_GetElementByTagName(item, "A");
    var nextItem, parentItem, previousItem;
    if ((!horizontal && key == 38) || (horizontal && key == 37)) {
        previousItem = Menu_FindPrevious(item);
        while (previousItem && previousItem.disabled) {
            previousItem = Menu_FindPrevious(previousItem);
        }
        if (previousItem) {
            Menu_Focus(previousItem);
            Menu_Expand(previousItem, data.horizontalOffset, data.verticalOffset, true);
            event.cancelBubble = true;
            if (event.stopPropagation) event.stopPropagation();
            return;
        }
    }
    if ((!horizontal && key == 40) || (horizontal && key == 39)) {
        if (horizontal) {
            var subMenu = Menu_FindSubMenu(a);
            if (subMenu && subMenu.style && subMenu.style.visibility &&
```

```
            subMenu.style.visibility.toLowerCase() == "hidden") {
            Menu_Expand(a, data.horizontalOffset, data.verticalOffset, true);
            event.cancelBubble = true;
            if (event.stopPropagation) event.stopPropagation();
            return;
        }
    }
    nextItem = Menu_FindNext(item);
    while (nextItem && nextItem.disabled) {
        nextItem = Menu_FindNext(nextItem);
    }
    if (nextItem) {
        Menu_Focus(nextItem);
        Menu_Expand(nextItem, data.horizontalOffset, data.verticalOffset, true);
        event.cancelBubble = true;
        if (event.stopPropagation) event.stopPropagation();
        return;
    }
}
if ((!horizontal && key == 39) || (horizontal && key == 40)) {
    var children = Menu_Expand(a, data.horizontalOffset, data.verticalOffset, true);
    if (children) {
        var firstChild;
        children = WebForm_GetElementsByTagName(children, "A");
        for (var i = 0; i < children.length; i++) {
            if (!children[i].disabled && Menu_IsSelectable(children[i])) {
                firstChild = children[i];
                break;
            }
        }
        if (firstChild) {
            Menu_Focus(firstChild);
            Menu_Expand(firstChild, data.horizontalOffset, data.verticalOffset, true);
            event.cancelBubble = true;
            if (event.stopPropagation) event.stopPropagation();
            return;
        }
    }
    else {
        parentItem = Menu_FindParentItem(item);
        while (parentItem && !Menu_IsHorizontal(parentItem)) {
            parentItem = Menu_FindParentItem(parentItem);
        }
        if (parentItem) {
            nextItem = Menu_FindNext(parentItem);
            while (nextItem && nextItem.disabled) {
                nextItem = Menu_FindNext(nextItem);
            }
            if (nextItem) {
                Menu_Focus(nextItem);
                Menu_Expand(nextItem, data.horizontalOffset, data.verticalOffset, true);
                event.cancelBubble = true;
                if (event.stopPropagation) event.stopPropagation();
                return;
            }
        }
    }
}
if ((!horizontal && key == 37) || (horizontal && key == 38)) {
    parentItem = Menu_FindParentItem(item);
```

```
        if (parentItem) {
            if (Menu_IsHorizontal(parentItem)) {
                previousItem = Menu_FindPrevious(parentItem);
                while (previousItem && previousItem.disabled) {
                    previousItem = Menu_FindPrevious(previousItem);
                }
                if (previousItem) {
                    Menu_Focus(previousItem);
                    Menu_Expand(previousItem, data.horizontalOffset, data.verticalOffset, true);
                    event.cancelBubble = true;
                    if (event.stopPropagation) event.stopPropagation();
                    return;
                }
            }
            var parentA = WebForm_GetElementByTagName(parentItem, "A");
            if (parentA) {
                Menu_Focus(parentA);
            }
            Menu_ResetSiblings(parentItem);
            event.cancelBubble = true;
            if (event.stopPropagation) event.stopPropagation();
            return;
        }
    }
    if (key == 27) {
        Menu_HideItems();
        event.cancelBubble = true;
        if (event.stopPropagation) event.stopPropagation();
        return;
    }
}
function Menu_ResetSiblings(item) {
    var table = (item.tagName.toLowerCase() == "td") ?
        item.parentNode.parentNode.parentNode :
        item.parentNode.parentNode;
    var isVertical = false;
    for (var r = 0; r < table.rows.length; r++) {
        if (table.rows[r].id) {
            isVertical = true;
            break;
        }
    }
    var i, child, childNode;
    if (isVertical) {
        for(i = 0; i < table.rows.length; i++) {
            childNode = table.rows[i];
            if (childNode != item) {
                child = WebForm_GetElementById(childNode.id + "Items");
                if (child) {
                    Menu_HideItems(child);
                }
            }
        }
    }
    else {
        for(i = 0; i < table.rows[0].cells.length; i++) {
            childNode = table.rows[0].cells[i];
            if (childNode != item) {
                child = WebForm_GetElementById(childNode.id + "Items");
                if (child) {
```

```
                Menu_HideItems(child);
            }
        }
    }
  }
  Menu_ResetTopMenus(table, table, 0, true);
}
function Menu_ResetTopMenus(table, doNotReset, level, up) {
  var i, child, childNode;
  if (up && table.id == "") {
    var parentTable = table.parentNode.parentNode.parentNode.parentNode;
    if (parentTable.tagName.toLowerCase() == "table") {
      Menu_ResetTopMenus(parentTable, doNotReset, level + 1, true);
    }
  }
  else {
    if (level == 0 && table != doNotReset) {
      if (table.rows[0].id) {
        for(i = 0; i < table.rows.length; i++) {
          childNode = table.rows[i];
          child = WebForm_GetElementById(childNode.id + "Items");
          if (child) {
            Menu_HideItems(child);
          }
        }
      }
      else {
        for(i = 0; i < table.rows[0].cells.length; i++) {
          childNode = table.rows[0].cells[i];
          child = WebForm_GetElementById(childNode.id + "Items");
          if (child) {
            Menu_HideItems(child);
          }
        }
      }
    }
    else if (level > 0) {
      for (i = 0; i < table.rows.length; i++) {
        for (var j = 0; j < table.rows[i].cells.length; j++) {
          var subTable = table.rows[i].cells[j].firstChild;
          if (subTable && subTable.tagName.toLowerCase() == "table") {
            Menu_ResetTopMenus(subTable, doNotReset, level - 1, false);
          }
        }
      }
    }
  }
}
function Menu_RestoreInterval() {
  if (__menuInterval && __rootMenuItem) {
    Menu_ClearInterval();
    __menuInterval = window.setInterval("Menu_HideItems()", __disappearAfter);
  }
}
function Menu_SetRoot(item) {
  var newRoot = Menu_FindMenu(item);
  if (newRoot) {
    if (__rootMenuItem && __rootMenuItem != newRoot) {
      Menu_HideItems();
    }
```

```
        __rootMenuItem = newRoot;
    }
}
function Menu_Unhover(item) {
    var node = (item.tagName.toLowerCase() == "td") ?
        item:
        item.cells[0];
    var nodeTable = WebForm_GetElementByTagName(node, "table");
    if (nodeTable.hoverClass) {
        WebForm_RemoveClassName(nodeTable, nodeTable.hoverClass);
    }
    node = nodeTable.rows[0].cells[0].childNodes[0];
    if (node.hoverHyperLinkClass) {
        WebForm_RemoveClassName(node, node.hoverHyperLinkClass);
    }
    Menu_Collapse(node);
}
function PopOut_Clip(element, y, height) {
    if (element && element.style) {
        element.style.clip = "rect(" + y + "px auto " + (y + height) + "px auto)";
        element.style.overflow = "hidden";
    }
}
function PopOut_Down(scroller) {
    Menu_ClearInterval();
    var panel;
    if (scroller) {
        panel = scroller.parentNode
    }
    else {
        panel = __scrollPanel;
    }
    if (panel && ((panel.offset + panel.clippedHeight) < panel.physicalHeight)) {
        PopOut_Scroll(panel, 2)
        __scrollPanel = panel;
        PopOut_ShowScrollers(panel);
        PopOut_Stop();
        __scrollPanel.interval = window.setInterval("PopOut_Down()", 8);
    }
    else {
        PopOut_ShowScrollers(panel);
    }
}
function PopOut_Hide(panelId) {
    var panel = WebForm_GetElementById(panelId);
    if (panel && panel.tagName.toLowerCase() == "div") {
        panel.style.visibility = "hidden";
        panel.style.display = "none";
        panel.offset = 0;
        panel.scrollTop = 0;
        var table = WebForm_GetElementByTagName(panel, "TABLE");
        if (table) {
            WebForm_SetElementY(table, 0);
        }
        if (window.navigator && window.navigator.appName == "Microsoft Internet Explorer" &&
            !window.opera) {
            var childFrameId = panel.id + "_MenuIFrame";
            var childFrame = WebForm_GetElementById(childFrameId);
            if (childFrame) {
                childFrame.style.display = "none";
```

```
            }
        }
    }
}
function PopOut_HideScrollers(panel) {
    if (panel && panel.style) {
        var up = WebForm_GetElementById(panel.id + "Up");
        var dn = WebForm_GetElementById(panel.id + "Dn");
        if (up) {
            up.style.visibility = "hidden";
            up.style.display = "none";
        }
        if (dn) {
            dn.style.visibility = "hidden";
            dn.style.display = "none";
        }
    }
}
function PopOut_Position(panel, hideScrollers) {
    if (window.opera) {
        panel.parentNode.removeChild(panel);
        document.forms[0].appendChild(panel);
    }
    var rel = WebForm_GetElementById(panel.rel);
    var relTable = WebForm_GetElementByTagName(rel, "TABLE");
    var relCoordinates = WebForm_GetElementPosition(relTable ? relTable : rel);
    var panelCoordinates = WebForm_GetElementPosition(panel);
    var panelHeight = ((typeof(panel.physicalHeight) != "undefined") && (panel.physicalHeight != null)) ?
        panel.physicalHeight :
        panelCoordinates.height;
    panel.physicalHeight = panelHeight;
    var panelParentCoordinates;
    if (panel.offsetParent) {
        panelParentCoordinates = WebForm_GetElementPosition(panel.offsetParent);
    }
    else {
        panelParentCoordinates = new Object();
        panelParentCoordinates.x = 0;
        panelParentCoordinates.y = 0;
    }
    var overflowElement = WebForm_GetElementById("__overFlowElement");
    if (!overflowElement) {
        overflowElement = document.createElement("img");
        overflowElement.id="__overFlowElement";
        WebForm_SetElementWidth(overflowElement, 1);
        document.body.appendChild(overflowElement);
    }
    WebForm_SetElementHeight(overflowElement, panelHeight + relCoordinates.y + parseInt(panel.y ? panel.y
: 0));
    overflowElement.style.visibility = "visible";
    overflowElement.style.display = "inline";
    var clientHeight = 0;
    var clientWidth = 0;
    if (window.innerHeight) {
        clientHeight = window.innerHeight;
        clientWidth = window.innerWidth;
    }
    else if (document.documentElement && document.documentElement.clientHeight) {
        clientHeight = document.documentElement.clientHeight;
        clientWidth = document.documentElement.clientWidth;
```

```
    }
    else if (document.body && document.body.clientHeight) {
        clientHeight = document.body.clientHeight;
        clientWidth = document.body.clientWidth;
    }
    var scrollTop = 0;
    var scrollLeft = 0;
    if (typeof(window.pageYOffset) != "undefined") {
        scrollTop = window.pageYOffset;
        scrollLeft = window.pageXOffset;
    }
    else if (document.documentElement && (typeof(document.documentElement.scrollTop) != "undefined")) {
        scrollTop = document.documentElement.scrollTop;
        scrollLeft = document.documentElement.scrollLeft;
    }
    else if (document.body && (typeof(document.body.scrollTop) != "undefined")) {
        scrollTop = document.body.scrollTop;
        scrollLeft = document.body.scrollLeft;
    }
    overflowElement.style.visibility = "hidden";
    overflowElement.style.display = "none";
    var bottomWindowBorder = clientHeight + scrollTop;
    var rightWindowBorder = clientWidth + scrollLeft;
    var position = panel.pos;
    if ((typeof(position) == "undefined") || (position == null) || (position == "")) {
        position = (WebForm_GetElementDir(rel) == "rtl" ? "middleleft" : "middleright");
    }
    position = position.toLowerCase();
    var y = relCoordinates.y + parseInt(panel.y ? panel.y : 0) - panelParentCoordinates.y;
    var borderParent = (rel && rel.parentNode && rel.parentNode.parentNode &&
rel.parentNode.parentNode.parentNode
        && rel.parentNode.parentNode.parentNode.tagName.toLowerCase() == "div") ?
        rel.parentNode.parentNode.parentNode : null;
    WebForm_SetElementY(panel, y);
    PopOut_SetPanelHeight(panel, panelHeight, true);
    var clip = false;
    var overflow;
    if (position.indexOf("top") != -1) {
        y -= panelHeight;
        WebForm_SetElementY(panel, y);
        if (y < -panelParentCoordinates.y) {
            y = -panelParentCoordinates.y;
            WebForm_SetElementY(panel, y);
            if (panelHeight > clientHeight - 2) {
                clip = true;
                PopOut_SetPanelHeight(panel, clientHeight - 2);
            }
        }
    }
    else {
        if (position.indexOf("bottom") != -1) {
            y += relCoordinates.height;
            WebForm_SetElementY(panel, y);
        }
        overflow = y + panelParentCoordinates.y + panelHeight - bottomWindowBorder;
        if (overflow > 0) {
            y -= overflow;
            WebForm_SetElementY(panel, y);
            if (y < -panelParentCoordinates.y) {
                y = 2 - panelParentCoordinates.y + scrollTop;
```

```
            WebForm_SetElementY(panel, y);
            clip = true;
            PopOut_SetPanelHeight(panel, clientHeight - 2);
         }
      }
   }
   if (!clip) {
      PopOut_SetPanelHeight(panel, panel.clippedHeight, true);
   }
   var panelParentOffsetY = 0;
   if (panel.offsetParent) {
      panelParentOffsetY = WebForm_GetElementPosition(panel.offsetParent).y;
   }
   var panelY = ((typeof(panel.originY) != "undefined") && (panel.originY != null)) ?
      panel.originY :
      y - panelParentOffsetY;
   panel.originY = panelY;
   if (!hideScrollers) {
      PopOut_ShowScrollers(panel);
   }
   else {
      PopOut_HideScrollers(panel);
   }
   var x = relCoordinates.x + parseInt(panel.x ? panel.x : 0) - panelParentCoordinates.x;
   if (borderParent && borderParent.clientLeft) {
      x += 2 * borderParent.clientLeft;
   }
   WebForm_SetElementX(panel, x);
   if (position.indexOf("left") != -1) {
      x -= panelCoordinates.width;
      WebForm_SetElementX(panel, x);
      if (x < -panelParentCoordinates.x) {
         WebForm_SetElementX(panel, -panelParentCoordinates.x);
      }
   }
   else {
      if (position.indexOf("right") != -1) {
         x += relCoordinates.width;
         WebForm_SetElementX(panel, x);
      }
      overflow = x + panelParentCoordinates.x + panelCoordinates.width - rightWindowBorder;
      if (overflow > 0) {
         if (position.indexOf("bottom") == -1 && relCoordinates.x > panelCoordinates.width) {
            x -= relCoordinates.width + panelCoordinates.width;
         }
         else {
            x -= overflow;
         }
         WebForm_SetElementX(panel, x);
         if (x < -panelParentCoordinates.x) {
            WebForm_SetElementX(panel, -panelParentCoordinates.x);
         }
      }
   }
}
function PopOut_Scroll(panel, offsetDelta) {
   var table = WebForm_GetElementByTagName(panel, "TABLE");
   if (!table) return;
   table.style.position = "relative";
   var tableY = (table.style.top ? parseInt(table.style.top) : 0);
```

```
    panel.offset += offsetDelta;
    WebForm_SetElementY(table, tableY - offsetDelta);
}
function PopOut_SetPanelHeight(element, height, doNotClip) {
    if (element && element.style) {
        var size = WebForm_GetElementPosition(element);
        element.physicalWidth = size.width;
        element.clippedHeight = height;
        WebForm_SetElementHeight(element, height - (element.clientTop ? (2 * element.clientTop) : 0));
        if (doNotClip && element.style) {
            element.style.clip = "rect(auto auto auto auto)";
        }
        else {
            PopOut_Clip(element, 0, height);
        }
    }
}
function PopOut_Show(panelId, hideScrollers, data) {
    var panel = WebForm_GetElementById(panelId);
    if (panel && panel.tagName.toLowerCase() == "div") {
        panel.style.visibility = "visible";
        panel.style.display = "inline";
        if (!panel.offset || hideScrollers) {
            panel.scrollTop = 0;
            panel.offset = 0;
            var table = WebForm_GetElementByTagName(panel, "TABLE");
            if (table) {
                WebForm_SetElementY(table, 0);
            }
        }
        PopOut_Position(panel, hideScrollers);
        var z = 1;
        var isIE = window.navigator && window.navigator.appName == "Microsoft Internet Explorer" &&
!window.opera;
        if (isIE && data) {
            var childFrameId = panel.id + "_MenuIFrame";
            var childFrame = WebForm_GetElementById(childFrameId);
            var parent = panel.offsetParent;
            if (!childFrame) {
                childFrame = document.createElement("iframe");
                childFrame.id = childFrameId;
                childFrame.src = (data.iframeUrl ? data.iframeUrl : "about:blank");
                childFrame.style.position = "absolute";
                childFrame.style.display = "none";
                childFrame.scrolling = "no";
                childFrame.frameBorder = "0";
                if (parent.tagName.toLowerCase() == "html") {
                    document.body.appendChild(childFrame);
                }
                else {
                    parent.appendChild(childFrame);
                }
            }
            var pos = WebForm_GetElementPosition(panel);
            var parentPos = WebForm_GetElementPosition(parent);
            WebForm_SetElementX(childFrame, pos.x - parentPos.x);
            WebForm_SetElementY(childFrame, pos.y - parentPos.y);
            WebForm_SetElementWidth(childFrame, pos.width);
            WebForm_SetElementHeight(childFrame, pos.height);
            childFrame.style.display = "block";
```

```
            if (panel.currentStyle && panel.currentStyle.zIndex) {
                z = panel.currentStyle.zIndex;
            }
            else if (panel.style.zIndex) {
                z = panel.style.zIndex;
            }
        }
        panel.style.zIndex = z;
    }
}
function PopOut_ShowScrollers(panel) {
    if (panel && panel.style) {
        var up = WebForm_GetElementById(panel.id + "Up");
        var dn = WebForm_GetElementById(panel.id + "Dn");
        var cnt = 0;
        if (up && dn) {
            if (panel.offset && panel.offset > 0) {
                up.style.visibility = "visible";
                up.style.display = "inline";
                cnt++;
                if (panel.clientWidth) {
                    WebForm_SetElementWidth(up, panel.clientWidth
                        - (up.clientLeft ? (2 * up.clientLeft) : 0));
                }
                WebForm_SetElementY(up, 0);
            }
            else {
                up.style.visibility = "hidden";
                up.style.display = "none";
            }
            if (panel.offset + panel.clippedHeight + 2 <= panel.physicalHeight) {
                dn.style.visibility = "visible";
                dn.style.display = "inline";
                cnt++;
                if (panel.clientWidth) {
                    WebForm_SetElementWidth(dn, panel.clientWidth
                        - (dn.clientLeft ? (2 * dn.clientLeft) : 0));
                }
                WebForm_SetElementY(dn, panel.clippedHeight - WebForm_GetElementPosition(dn).height
                    - (panel.clientTop ? (2 * panel.clientTop) : 0));
            }
            else {
                dn.style.visibility = "hidden";
                dn.style.display = "none";
            }
            if (cnt == 0) {
                panel.style.clip = "rect(auto auto auto auto)";
            }
        }
    }
}
function PopOut_Stop() {
    if (__scrollPanel && __scrollPanel.interval) {
        window.clearInterval(__scrollPanel.interval);
    }
    Menu_RestoreInterval();
}
function PopOut_Up(scroller) {
    Menu_ClearInterval();
    var panel;
```

```
    if (scroller) {
        panel = scroller.parentNode
    }
    else {
        panel = __scrollPanel;
    }
    if (panel && panel.offset && panel.offset > 0) {
        PopOut_Scroll(panel, -2);
        __scrollPanel = panel;
        PopOut_ShowScrollers(panel);
        PopOut_Stop();
        __scrollPanel.interval = window.setInterval("PopOut_Up()", 8);
    }
}
```

Famille de fonctions javascript de gestion d'une fiche

```
function WebForm_PostBackOptions(eventTarget, eventArgument, validation, validationGroup, actionUrl,
trackFocus, clientSubmit) {
    this.eventTarget = eventTarget;
    this.eventArgument = eventArgument;
    this.validation = validation;
    this.validationGroup = validationGroup;
    this.actionUrl = actionUrl;
    this.trackFocus = trackFocus;
    this.clientSubmit = clientSubmit;
}
function WebForm_DoPostBackWithOptions(options) {
    var validationResult = true;
    if (options.validation) {
        if (typeof(Page_ClientValidate) == 'function') {
            validationResult = Page_ClientValidate(options.validationGroup);
        }
    }
    if (validationResult) {
        if ((typeof(options.actionUrl) != "undefined") && (options.actionUrl != null) && (options.actionUrl.length >
0)) {
            theForm.action = options.actionUrl;
        }
        if (options.trackFocus) {
            var lastFocus = theForm.elements["__LASTFOCUS"];
            if ((typeof(lastFocus) != "undefined") && (lastFocus != null)) {
                if (typeof(document.activeElement) == "undefined") {
                    lastFocus.value = options.eventTarget;
                }
                else {
                    var active = document.activeElement;
                    if ((typeof(active) != "undefined") && (active != null)) {
                        if ((typeof(active.id) != "undefined") && (active.id != null) && (active.id.length > 0)) {
                            lastFocus.value = active.id;
                        }
                        else if (typeof(active.name) != "undefined") {
                            lastFocus.value = active.name;
                        }
                    }
                }
            }
        }
```

```
    }
    if (options.clientSubmit) {
        __doPostBack(options.eventTarget, options.eventArgument);
    }
}
var __pendingCallbacks = new Array();
var __synchronousCallBackIndex = -1;
function WebForm_DoCallback(eventTarget, eventArgument, eventCallback, context, errorCallback, useAsync) {
    var postData = __theFormPostData +
            "__CALLBACKID=" + WebForm_EncodeCallback(eventTarget) +
            "&__CALLBACKPARAM=" + WebForm_EncodeCallback(eventArgument);
    if (theForm["__EVENTVALIDATION"]) {
        postData += "&__EVENTVALIDATION=" +
WebForm_EncodeCallback(theForm["__EVENTVALIDATION"].value);
    }
    var xmlRequest,e;
    try {
        xmlRequest = new XMLHttpRequest();
    }
    catch(e) {
        try {
            xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch(e) {
        }
    }
    var setRequestHeaderMethodExists = true;
    try {
        setRequestHeaderMethodExists = (xmlRequest && xmlRequest.setRequestHeader);
    }
    catch(e) {}
    var callback = new Object();
    callback.eventCallback = eventCallback;
    callback.context = context;
    callback.errorCallback = errorCallback;
    callback.async = useAsync;
    var callbackIndex = WebForm_FillFirstAvailableSlot(__pendingCallbacks, callback);
    if (!useAsync) {
        if (__synchronousCallBackIndex != -1) {
            __pendingCallbacks[__synchronousCallBackIndex] = null;
        }
        __synchronousCallBackIndex = callbackIndex;
    }
    if (setRequestHeaderMethodExists) {
        xmlRequest.onreadystatechange = WebForm_CallbackComplete;
        callback.xmlRequest = xmlRequest;
        xmlRequest.open("POST", theForm.action, true);
        xmlRequest.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
        xmlRequest.send(postData);
        return;
    }
    callback.xmlRequest = new Object();
    var callbackFrameID = "__CALLBACKFRAME" + callbackIndex;
    var xmlRequestFrame = document.frames[callbackFrameID];
    if (!xmlRequestFrame) {
        xmlRequestFrame = document.createElement("IFRAME");
        xmlRequestFrame.width = "1";
        xmlRequestFrame.height = "1";
        xmlRequestFrame.frameBorder = "0";
        xmlRequestFrame.id = callbackFrameID;
```

```
        xmlRequestFrame.name = callbackFrameID;
        xmlRequestFrame.style.position = "absolute";
        xmlRequestFrame.style.top = "-100px"
        xmlRequestFrame.style.left = "-100px";
        try {
            if (callBackFrameUrl) {
                xmlRequestFrame.src = callBackFrameUrl;
            }
        }
        catch(e) {}
        document.body.appendChild(xmlRequestFrame);
    }
    var interval = window.setInterval(function() {
        xmlRequestFrame = document.frames[callbackFrameID];
        if (xmlRequestFrame && xmlRequestFrame.document) {
            window.clearInterval(interval);
            xmlRequestFrame.document.write("");
            xmlRequestFrame.document.close();
            xmlRequestFrame.document.write('<html><body><form method="post"><input type="hidden"
name="__CALLBACKLOADSCRIPT" value="t"></form></body></html>');
            xmlRequestFrame.document.close();
            xmlRequestFrame.document.forms[0].action = theForm.action;
            var count = __theFormPostCollection.length;
            var element;
            for (var i = 0; i < count; i++) {
                element = __theFormPostCollection[i];
                if (element) {
                    var fieldElement = xmlRequestFrame.document.createElement("INPUT");
                    fieldElement.type = "hidden";
                    fieldElement.name = element.name;
                    fieldElement.value = element.value;
                    xmlRequestFrame.document.forms[0].appendChild(fieldElement);
                }
            }
            var callbackIdFieldElement = xmlRequestFrame.document.createElement("INPUT");
            callbackIdFieldElement.type = "hidden";
            callbackIdFieldElement.name = "__CALLBACKID";
            callbackIdFieldElement.value = eventTarget;
            xmlRequestFrame.document.forms[0].appendChild(callbackIdFieldElement);
            var callbackParamFieldElement = xmlRequestFrame.document.createElement("INPUT");
            callbackParamFieldElement.type = "hidden";
            callbackParamFieldElement.name = "__CALLBACKPARAM";
            callbackParamFieldElement.value = eventArgument;
            xmlRequestFrame.document.forms[0].appendChild(callbackParamFieldElement);
            if (theForm["__EVENTVALIDATION"]) {
                var callbackValidationFieldElement = xmlRequestFrame.document.createElement("INPUT");
                callbackValidationFieldElement.type = "hidden";
                callbackValidationFieldElement.name = "__EVENTVALIDATION";
                callbackValidationFieldElement.value = theForm["__EVENTVALIDATION"].value;
                xmlRequestFrame.document.forms[0].appendChild(callbackValidationFieldElement);
            }
            var callbackIndexFieldElement = xmlRequestFrame.document.createElement("INPUT");
            callbackIndexFieldElement.type = "hidden";
            callbackIndexFieldElement.name = "__CALLBACKINDEX";
            callbackIndexFieldElement.value = callbackIndex;
            xmlRequestFrame.document.forms[0].appendChild(callbackIndexFieldElement);
            xmlRequestFrame.document.forms[0].submit();
        }
    }, 10);
}
```

```
function WebForm_CallbackComplete() {
   for (i = 0; i < __pendingCallbacks.length; i++) {
      callbackObject = __pendingCallbacks[i];
      if (callbackObject && callbackObject.xmlRequest && (callbackObject.xmlRequest.readyState == 4)) {
         WebForm_ExecuteCallback(callbackObject);
         if (!__pendingCallbacks[i].async) {
            __synchronousCallBackIndex = -1;
         }
         __pendingCallbacks[i] = null;
         var callbackFrameID = "__CALLBACKFRAME" + i;
         var xmlRequestFrame = document.getElementById(callbackFrameID);
         if (xmlRequestFrame) {
            xmlRequestFrame.parentNode.removeChild(xmlRequestFrame);
         }
      }
   }
}
function WebForm_ExecuteCallback(callbackObject) {
   var response = callbackObject.xmlRequest.responseText;
   if (response.charAt(0) == "s") {
      if ((typeof(callbackObject.eventCallback) != "undefined") && (callbackObject.eventCallback != null)) {
         callbackObject.eventCallback(response.substring(1), callbackObject.context);
      }
   }
   else if (response.charAt(0) == "e") {
      if ((typeof(callbackObject.errorCallback) != "undefined") && (callbackObject.errorCallback != null)) {
         callbackObject.errorCallback(response.substring(1), callbackObject.context);
      }
   }
   else {
      var separatorIndex = response.indexOf("|");
      if (separatorIndex != -1) {
         var validationFieldLength = parseInt(response.substring(0, separatorIndex));
         if (!isNaN(validationFieldLength)) {
            var validationField = response.substring(separatorIndex + 1, separatorIndex + validationFieldLength + 1);
            if (validationField != "") {
               var validationFieldElement = theForm["__EVENTVALIDATION"];
               if (!validationFieldElement) {
                  validationFieldElement = document.createElement("INPUT");
                  validationFieldElement.type = "hidden";
                  validationFieldElement.name = "__EVENTVALIDATION";
                  theForm.appendChild(validationFieldElement);
               }
               validationFieldElement.value = validationField;
            }
            if ((typeof(callbackObject.eventCallback) != "undefined") && (callbackObject.eventCallback != null)) {
               callbackObject.eventCallback(response.substring(separatorIndex + validationFieldLength + 1),
callbackObject.context);
            }
         }
      }
   }
}
function WebForm_FillFirstAvailableSlot(array, element) {
   var i;
   for (i = 0; i < array.length; i++) {
      if (!array[i]) break;
   }
   array[i] = element;
   return i;
```

```
}
var __nonMSDOMBrowser = (window.navigator.appName.toLowerCase().indexOf('explorer') == -1);
var __theFormPostData = "";
var __theFormPostCollection = new Array();
function WebForm_InitCallback() {
   var count = theForm.elements.length;
   var element;
   for (var i = 0; i < count; i++) {
      element = theForm.elements[i];
      var tagName = element.tagName.toLowerCase();
      if (tagName == "input") {
         var type = element.type;
         if ((type == "text" || type == "hidden" || type == "password" ||
            ((type == "checkbox" || type == "radio") && element.checked)) &&
            (element.id != "__EVENTVALIDATION")) {
            WebForm_InitCallbackAddField(element.name, element.value);
         }
      }
      else if (tagName == "select") {
         var selectCount = element.options.length;
         for (var j = 0; j < selectCount; j++) {
            var selectChild = element.options[j];
            if (selectChild.selected == true) {
               WebForm_InitCallbackAddField(element.name, element.value);
            }
         }
      }
      else if (tagName == "textarea") {
         WebForm_InitCallbackAddField(element.name, element.value);
      }
   }
}
function WebForm_InitCallbackAddField(name, value) {
   var nameValue = new Object();
   nameValue.name = name;
   nameValue.value = value;
   __theFormPostCollection[__theFormPostCollection.length] = nameValue;
   __theFormPostData += name + "=" + WebForm_EncodeCallback(value) + "&";
}
function WebForm_EncodeCallback(parameter) {
   if (encodeURIComponent) {
      return encodeURIComponent(parameter);
   }
   else {
      return escape(parameter);
   }
}
var __disabledControlArray = new Array();
function WebForm_ReEnableControls() {
   if (typeof(__enabledControlArray) == 'undefined') {
      return false;
   }
   var disabledIndex = 0;
   for (var i = 0; i < __enabledControlArray.length; i++) {
      var c;
      if (__nonMSDOMBrowser) {
         c = document.getElementById(__enabledControlArray[i]);
      }
      else {
         c = document.all[__enabledControlArray[i]];
```

```
        }
        if ((typeof(c) != "undefined") && (c != null) && (c.disabled == true)) {
            c.disabled = false;
            __disabledControlArray[disabledIndex++] = c;
        }
    }
    setTimeout("WebForm_ReDisableControls()", 0);
    return true;
}
function WebForm_ReDisableControls() {
    for (var i = 0; i < __disabledControlArray.length; i++) {
        __disabledControlArray[i].disabled = true;
    }
}
var __defaultFired = false;
function WebForm_FireDefaultButton(event, target) {
    if (!__defaultFired && event.keyCode == 13 && !(event.srcElement &&
(event.srcElement.tagName.toLowerCase() == "textarea"))) {
        var defaultButton;
        if (__nonMSDOMBrowser) {
            defaultButton = document.getElementById(target);
        }
        else {
            defaultButton = document.all[target];
        }
        if (defaultButton && typeof(defaultButton.click) != "undefined") {
            __defaultFired = true;
            defaultButton.click();
            event.cancelBubble = true;
            if (event.stopPropagation) event.stopPropagation();
            return false;
        }
    }
    return true;
}
function WebForm_GetScrollX() {
    if (__nonMSDOMBrowser) {
        return window.pageXOffset;
    }
    else {
        if (document.documentElement && document.documentElement.scrollLeft) {
            return document.documentElement.scrollLeft;
        }
        else if (document.body) {
            return document.body.scrollLeft;
        }
    }
    return 0;
}
function WebForm_GetScrollY() {
    if (__nonMSDOMBrowser) {
        return window.pageYOffset;
    }
    else {
        if (document.documentElement && document.documentElement.scrollTop) {
            return document.documentElement.scrollTop;
        }
        else if (document.body) {
            return document.body.scrollTop;
        }
```

```
    }
    return 0;
}
function WebForm_SaveScrollPositionSubmit() {
    if (__nonMSDOMBrowser) {
        theForm.elements['__SCROLLPOSITIONY'].value = window.pageYOffset;
        theForm.elements['__SCROLLPOSITIONX'].value = window.pageXOffset;
    }
    else {
        theForm.__SCROLLPOSITIONX.value = WebForm_GetScrollX();
        theForm.__SCROLLPOSITIONY.value = WebForm_GetScrollY();
    }
    if ((typeof(this.oldSubmit) != "undefined") && (this.oldSubmit != null)) {
        return this.oldSubmit();
    }
    return true;
}
function WebForm_SaveScrollPositionOnSubmit() {
    theForm.__SCROLLPOSITIONX.value = WebForm_GetScrollX();
    theForm.__SCROLLPOSITIONY.value = WebForm_GetScrollY();
    if ((typeof(this.oldOnSubmit) != "undefined") && (this.oldOnSubmit != null)) {
        return this.oldOnSubmit();
    }
    return true;
}
function WebForm_RestoreScrollPosition() {
    if (__nonMSDOMBrowser) {
        window.scrollTo(theForm.elements['__SCROLLPOSITIONX'].value,
theForm.elements['__SCROLLPOSITIONY'].value);
    }
    else {
        window.scrollTo(theForm.__SCROLLPOSITIONX.value, theForm.__SCROLLPOSITIONY.value);
    }
    if ((typeof(theForm.oldOnLoad) != "undefined") && (theForm.oldOnLoad != null)) {
        return theForm.oldOnLoad();
    }
    return true;
}
function WebForm_TextBoxKeyHandler(event) {
    if (event.keyCode == 13) {
        var target;
        if (__nonMSDOMBrowser) {
            target = event.target;
        }
        else {
            target = event.srcElement;
        }
        if ((typeof(target) != "undefined") && (target != null)) {
            if (typeof(target.onchange) != "undefined") {
                target.onchange();
                event.cancelBubble = true;
                if (event.stopPropagation) event.stopPropagation();
                return false;
            }
        }
    }
    return true;
}
function WebForm_AppendToClassName(element, className) {
    var current = element.className;
```

```
    if (current) {
        if (current.charAt(current.length - 1) != ' ') {
            current += ' ';
        }
        current += className;
    }
    else {
        current = className;
    }
    element.className = current;
}
function WebForm_RemoveClassName(element, className) {
    var current = element.className;
    if (current) {
        if (current.substring(current.length - className.length - 1, current.length) == ' ' + className) {
            element.className = current.substring(0, current.length - className.length - 1);
            return;
        }
        if (current == className) {
            element.className = "";
            return;
        }
        var index = current.indexOf(' ' + className + ' ');
        if (index != -1) {
            element.className = current.substring(0, index) + current.substring(index + className.length + 2,
current.length);
            return;
        }
        if (current.substring(0, className.length) == className + ' ') {
            element.className = current.substring(className.length + 1, current.length);
        }
    }
}
function WebForm_GetElementById(elementId) {
    if (document.getElementById) {
        return document.getElementById(elementId);
    }
    else if (document.all) {
        return document.all[elementId];
    }
    else return null;
}
function WebForm_GetElementByTagName(element, tagName) {
    var elements = WebForm_GetElementsByTagName(element, tagName);
    if (elements && elements.length > 0) {
        return elements[0];
    }
    else return null;
}
function WebForm_GetElementsByTagName(element, tagName) {
    if (element && tagName) {
        if (element.getElementsByTagName) {
            return element.getElementsByTagName(tagName);
        }
        if (element.all && element.all.tags) {
            return element.all.tags(tagName);
        }
    }
    return null;
}
```

```
function WebForm_GetElementDir(element) {
   if (element) {
      if (element.dir) {
         return element.dir;
      }
      return WebForm_GetElementDir(element.parentNode);
   }
   return "ltr";
}
function WebForm_GetElementPosition(element) {
   var result = new Object();
   result.x = 0;
   result.y = 0;
   result.width = 0;
   result.height = 0;
   if (element.offsetParent) {
      result.x = element.offsetLeft;
      result.y = element.offsetTop;
      var parent = element.offsetParent;
      while (parent) {
         result.x += parent.offsetLeft;
         result.y += parent.offsetTop;
         var parentTagName = parent.tagName.toLowerCase();
         if (parentTagName != "table" &&
            parentTagName != "body" &&
            parentTagName != "html" &&
            parentTagName != "div" &&
            parent.clientTop &&
            parent.clientLeft) {
            result.x += parent.clientLeft;
            result.y += parent.clientTop;
         }
         parent = parent.offsetParent;
      }
   }
   else if (element.left && element.top) {
      result.x = element.left;
      result.y = element.top;
   }
   else {
      if (element.x) {
         result.x = element.x;
      }
      if (element.y) {
         result.y = element.y;
      }
   }
   if (element.offsetWidth && element.offsetHeight) {
      result.width = element.offsetWidth;
      result.height = element.offsetHeight;
   }
   else if (element.style && element.style.pixelWidth && element.style.pixelHeight) {
      result.width = element.style.pixelWidth;
      result.height = element.style.pixelHeight;
   }
   return result;
}
function WebForm_GetParentByTagName(element, tagName) {
   var parent = element.parentNode;
   var upperTagName = tagName.toUpperCase();
```

```
    while (parent && (parent.tagName.toUpperCase() != upperTagName)) {
        parent = parent.parentNode ? parent.parentNode : parent.parentElement;
    }
    return parent;
}
function WebForm_SetElementHeight(element, height) {
    if (element && element.style) {
        element.style.height = height + "px";
    }
}
function WebForm_SetElementWidth(element, width) {
    if (element && element.style) {
        element.style.width = width + "px";
    }
}
function WebForm_SetElementX(element, x) {
    if (element && element.style) {
        element.style.left = x + "px";
    }
}
function WebForm_SetElementY(element, y) {
    if (element && element.style) {
        element.style.top = y + "px";
    }
}
```

Exercices

Algorithmes simples avec

Calcul de la valeur absolue d'un nombre réel ……..p. 4
Résolution de l'équation du second degré dans R ……...……..p. 6
Calcul des nombres de Armstrong ……...……..p. 8
Calcul de nombres parfaits ……...……...……..p. 10
Calcul du pgcd de 2 entiers (méthode Euclide) ……...……..p. 12
Calcul du pgcd de 2 entiers (méthode Egyptienne) ……..…...……..p. 14
Calcul de nombres premiers (boucles **while** et **do**…**while**) **……...……..p. 16**
Calcul de nombres premiers (boucles **for**) **……...……..p. 18**
Calcul du nombre d'or ……...……..p. 20
Conjecture de Goldbach ……..……...……..p. 22
Méthodes d'opérations sur 8 bits ……...……...……..p. 24
Chaînes palindromes 2versions ……...……...……..p. 28
Convertir une date numérique en lettres ……...……...……..p. 31
Convertir un nombre écrit en chiffres romains ……...……...……..p. 33
Tri à bulles tableau d'entiers ……...……..…...……..p. 34
Tri par insertion tableau d'entiers ……..…...……..p. 36
Recherche linéaire dans un tableau non trié ……..…...……..p. 39
Recherche linéaire dans un tableau déjà trié ……...……..p. 43
Recherche dichotomique dans un tableau déjà trié ……...……..p. 46

*Dans la majorité des exemples de traduction d'algorithmes simples nous reprenons volontairement des exemples déjà traités dans un autre livre pdf, les **fondements de Java**, le lecteur remarquera la similarité du code C# et du code Java pour le même exercice.*

Lorsqu'il y a dissemblance du code C# avec le code Java, nous avons fourni des explications spécifiques au code C#.

Classes, objets et IHM avec

Problème de la référence circulaire……..……..p. 49
Classe de salariés dans une entreprise fictive ……..p. 51
Classe de salariés dans un fichier de l'entreprise ……..……...p. 62
Construction d'un ensemble de caractères ……...……..p. 70
Construction d'un ensemble générique ……..……...……..p. 76
Construction d'une IHM de jeu de puzzle …..……...…….p. 84

Les exercices utilisent des classes spécifiques au langage C# dans .Net, le lecteur trouvera plus d'exercices avec leur corrections sur la POO et C# dans l'exerciseur interactif construit par l'auteur et téléchargeable à :

http://www.discala.net

Algorithme

Calcul de la valeur absolue d'un nombre réel

Objectif : Ecrire un programme C# servant à calculer la valeur absolue d'un nombre réel x à partir de la définition de la valeur absolue. La valeur absolue du nombre réel x est le nombre réel |x| :

$$|x| = x \text{ , si } x \geq 0$$

$$|x| = -x \text{ si } x < 0$$

Spécifications de l'algorithme :

```
lire( x );
si x ≥ 0 alors écrire( '|x| =', x)
sinon écrire( '|x| =', -x)
fsi
```

Implantation en C#

Ecrivez avec les deux instructions différentes "**if...else..**" et "**...?.. : ...**", le programme C# complet correspondant à l'affichage ci-dessous :

```
Entrez un nombre x = -45
  |x| = 45
```

Proposition de squelette de classe C# à implanter :

```
class ApplicationValAbsolue {
    static void Main(string[ ] args) {
        ……..
    }
  }
```

La méthode **Main** calcule et affiche la valeur absolue.

Classe C# solution

Une classe C# solution du problème avec un if...else :

```
using System;
namespace CsExosAlgo1 {
class  ApplicationValAbsolue1 {
   static void Main(string[ ] args) {
        double x;
        System.Console.Write("Entrez un nombre x = ");
        x = Double.Parse( System.Console.ReadLine( ) ) ;
        if (x<0) System.Console.WriteLine("|x| = "+(-x));
        else System.Console.WriteLine("|x| = "+x);
   }
 }
}
```

Explication sur l'instruction :

```
x = Double.Parse ( System.Console.ReadLine( ) ) ;
```

Le gestionnaire d'entrée sortie C# à partir de la classe Console renvoie à travers la méthode ReadLine() une valeur saisie au clavier de type string. Il est donc obligatoire si l'on veut récuperer un nombre réel au clavier (ici **double** x;) de transtyper le réel tapé correctement sous forme de chaîne au clavier, et de le convertir en un réel de type **double** ici, grâce à la méthode Parse de la classe enveloppe Double du type **double**.

Remarquons que cette version simple ne protège pas des erreurs de saisie. Pour être plus robuste le programme devrait intercepter l'exception levée par une éventuelle erreur de saisie signalée par une exception du type FormatException :

```
try {
  x = Double.Parse( System.Console.ReadLine( ) ) ;
}
catch ( FormatException ) {  //...traitement de l'erreur de saisie }
```

Une classe C# solution du problème avec un "... ? ... : ..." :

```
using System;
namespace CsExosAlgo1
{
class ApplicationValAbsolue2 {
static void Main(string[ ] args) {
        double x;
        System.Console.Write("Entrez un nombre x = ");
        x = Double.Parse( System.Console.ReadLine( ) ) ;
        System.Console.WriteLine("|x| = "+ (x < 0 ? -x : x) );
  }
}
}
```

Algorithme

Algorithme de résolution de l'équation du second degré dans R.

Objectif : On souhaite écrire un programme C# de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$

Il s'agit ici d'un algorithme très classique provenant du cours de mathématique des classes du secondaire. L'exercice consiste essentiellement en la traduction immédiate

Spécifications de l'algorithme :

```
Algorithme Equation
   Entrée: A, B, C ∈ Réels
   Sortie: X1 , X2  ∈ Réels
   Local:  Δ ∈ Réels

début
  lire(A, B, C);
  Si A=0 alors début{A=0}
    Si B = 0 alors
      Si C = 0 alors
        écrire(R est solution)
      Sinon{C ≠ 0}
        écrire(pas de solution)
      Fsi
    Sinon {B ≠ 0}
      X1 ← C/B;
      écrire (X1)
    Fsi
  fin
  Sinon {A ≠ 0}début
    Δ ← B² - 4*A*C ;
      Si  Δ < 0  alors
        écrire(pas de solution)
      Sinon {Δ ≥ 0}
        Si Δ = 0 alors
          X1 ← -B/(2*A);
          écrire (X1)
        Sinon{Δ ≠ 0}
          X1 ←  (-B +√Δ )/(2*A);
          X2 ← (-B - √Δ)/(2*A);
          écrire(X1 , X2 )
        Fsi
      Fsi
    fin
    Fsi
  FinEquation
```

Implantation en C#

Ecrivez le programme C# qui est la traduction immédiate de cet algorithme dans le corps de la méthode Main.

Proposition de squelette de classe C# à implanter :

```
class ApplicationEqua2 {
    static void Main(string[ ] args) {
        ……..
    }
  }
```

Conseil :

On utilisera la méthode **static** Sqrt(double x) de la classe **Math** pour calculer la racine carrée d'un nombre réel :

$\sqrt{\Delta}$ se traduira alors par : Math.Sqrt(delta)

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
class ApplicationEqua2 {
 static void Main (string[ ] arg)   {
  double a, b, c, delta ;
  double x, x1, x2 ;
  System.Console.Write("Entrer une valeur pour a : ") ;
  a = Double.Parse( System.Console.ReadLine( ) ) ;
  System.Console.Write("Entrer une valeur pour b : ") ;
  b = Double.Parse( System.Console.ReadLine( ) ) ;
  System.Console.Write("Entrer une valeur pour c : ") ;
  c = Double.Parse( System.Console.ReadLine( ) ) ;
  if (a ==0)  {
   if (b ==0)  {
    if (c ==0)   {
      System.Console.WriteLine("tout reel est solution") ;
    }
    else  {// c ≠ 0
      System.Console.WriteLine("il n'y a pas de solution") ;
    }
   }
   else  { // b ≠ 0
      x = -c/b ;
      System.Console.WriteLine("la solution est " + x) ;
   }
  }
  else  { // a ≠ 0
   delta = b*b - 4*a*c ;
   if (delta < 0)    {
     System.Console.WriteLine("il n'y a pas de solution dans les reels") ;
   }
   else  { // delta  ≥ 0
     x1 = (-b + Math.Sqrt(delta))/ (2*a) ;
     x2 = (-b - Math.Sqrt(delta))/ (2*a) ;
     System.Console.WriteLine("il y deux solutions egales a " + x1 + " et " + x2) ;
   }
  }
 }
}
}
```

Algorithme

Calcul des nombres de Armstrong

Objectif : On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent.

***Exemple* :**

$153 = 1^3 + 5^3 + 3^3$

153 = 1 + 125 + 27, est un nombre de Armstrong.

Spécifications de l'algorithme :

On sait qu'il n'existe que 4 nombres de Armstrong, et qu'ils ont tous 3 chiffres (ils sont compris entre 100 et 500).
Si l'on suppose qu'un tel nombre est écrit **ijk** (**i** chiffre des centaines, **j** chiffres des dizaines et **k** chiffres des unités), il suffit simplement d'envisager tous les nombres possibles en faisant varier les chiffres entre 0 et 9 et de tester si le nombre est de Armstrong.

Implantation en C#

Ecrivez le programme C# complet qui fournisse les 4 nombres de Armstrong :

```
Nombres de Armstrong:
   153
   370
   371
   407
```

Proposition de squelette de classe C# à implanter :

```
class ApplicationArmstrong {
    static void Main(string[ ] args) {
        ……..
    }
  }
```

La méthode **Main** calcule et affiche les nombres de Armstrong.

Squelette plus détaillé de la classe C# à implanter :

```
using System;
namespace CsExosAlgo1
{
class ApplicationArmstrong {
static void Main(string[ ] args) {
        int i, j, k, n, somcube;
        System.Console.WriteLine("Nombres de Armstrong:");
        for(i = 1; i<=9; i++)
          for(j = 0; j<=9; j++)
           for(k = 0; k<=9; k++)
           {
                    ……
           }
  }
}
}
```

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
  class ApplicationArmstrong {
    static void Main(string[ ] args) {
        int i, j, k, n, somcube;
        System.Console.WriteLine("Nombres de Armstrong:");
        for(i = 1; i<=9; i++)
          for(j = 0; j<=9; j++)
           for(k = 0; k<=9; k++)
           {
                n = 100*i + 10*j + k;
               somcube = i*i*i + j*j*j + k*k*k;
               if (somcube == n)
                  System.Console.WriteLine(n);
           }
    }
  }
}
```

Algorithme

Calcul de nombres parfaits

Objectif : On souhaite écrire un programme C# de calcul des n premiers nombres parfaits. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs, 1 compris.

***Exemple* : 6 = 1+2+3 , est un nombre parfait.**

Spécifications de l'algorithme :

l'algorithme retenu contiendra deux boucles imbriquées. Une boucle de comptage des nombres parfaits qui s'arrêtera lorsque le décompte sera atteint, la boucle interne ayant vocation à calculer tous les diviseurs du nombre examiné d'en faire la somme puis de tester l'égalité entre cette somme et le nombre.

```
Algorithme Parfait
  Entrée: n  ∈ N
  Sortie: nbr ∈ N
  Local: somdiv, k, compt   ∈ N

     début
       lire(n);
       compt  ← 0;
       nbr  ←2;
      Tantque(compt < n) Faire
        somdiv ← 1;
        Pour k ← 2 jusquà nbr-1 Faire
          Si reste(nbr par k) = 0 Alors // k  divise nbr
            somdiv ← somdiv + k
          Fsi
        Fpour ;
        Si somdiv = nbr Alors
          ecrire(nbr) ;
          compt ← compt+1;
        Fsi;
        nbr ← nbr+1
      Ftant

FinParfait
```

Implantation en C#

Ecrivez le programme C# complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

```
Entrez combien de nombre parfaits : 4
    6 est un nombre parfait
    28 est un nombre parfait
    496 est un nombre parfait
    8128 est un nombre parfait
```

Proposition de squelette de classe C# à implanter :

```
class ApplicationParfaits {
    static void Main(string[ ] args) {
        ........
    }
  }
```

La méthode **Main** calcule et affiche les nombres parfaits.

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
  class ApplicationParfaits {
    static void Main (string[ ] args) {
        int compt = 0, n, k, somdiv, nbr;
        System.Console.Write("Entrez combien de nombre parfaits : ");
        n  = Int32.Parse( System.Console.ReadLine( ) ) ;
        nbr = 2;
        while (compt != n)
        { somdiv = 1;
          k = 2;
         while(k <= nbr/2 )
         {
         if (nbr % k == 0) somdiv += k ;
            k++;
         }
         if (somdiv == nbr)
         { System.Console.WriteLine(nbr+" est un nombre parfait");
            compt++;
         }
          nbr++;
        }
    }
  }
}
```

La saisie de l'entier **int** n; s'effectue par transtypage grâce à la méthode Parse de la classe Net Framework Int32 du type **int**.

Algorithme

Calcul du pgcd de 2 entiers (méthode Euclide)

Objectif : On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en C# à partir de l'algorithme de la méthode d'Euclide. Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) a et b, selon cette méthode :

Spécifications de l'algorithme :

```
Algorithme Pgcd
 Entrée: a,b  ∈  N* x N*
 Sortie: pgcd  ∈ N
 Local: r,t  ∈ N x N

début
  lire(a,b);
  Si ba Alors
    t ← a ;
    a ← b ;
    b ← t
  Fsi;
  Répéter
    r ← a mod b ;
    a ← b ;
    b ← r
  jusquà r = 0;
  pgcd ← a;
  ecrire(pgcd)
FinPgcd
```

Implantation en C#

Ecrivez le programme C# complet qui produise le dialogue suivant à la console (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

Proposition de squelette de classe C# à implanter :

```
class ApplicationEuclide {
   static void Main(string[ ] args) {
      ……..
    }
    static int pgcd (int a, int b) {
      ……..
    }
  }
```

La méthode **pgcd** renvoie le pgcd des deux entiers p et q .

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
class ApplicationEuclide {
 static void Main (string[ ] args) {
      System.Console.Write("Entrez le premier nombre : ");
     int p  = Int32.Parse( System.Console.ReadLine( ) ) ;
     System.Console.Write("Entrez le deuxième nombre : ");
     int q  = Int32.Parse( System.Console.ReadLine( ) ) ;
     if (p*q!=0)
       System.Console.WriteLine("Le pgcd de "+p+" et de "+q+" est "+pgcd(p,q));
     else
     System.Console.WriteLine("Le pgcd n'existe pas lorsque l'un des deux nombres est nul !");
  }

 static int pgcd (int a , int b) {
   int r,t ;
   if ( b>a) {
          t = a;
          a = b;
          b = t;
    }
    do {
          r = a % b;
          a = b;
          b = r;
   } while(r !=0);
   return a ;
  }
  }
 }
```

Algorithme

Calcul du pgcd de 2 entiers (méthode Egyptienne)

Objectif : On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en C# à partir de l'algorithme de la méthode dite "égyptienne " Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) p et q, selon cette méthode :

Spécifications de l'algorithme :

```
Lire (p, q ) ;

Tantque p ≠ q faire
   Si p > q alors

      p ← p – q

    sinon

      q ← q – p

    FinSi
 FinTant;
 Ecrire(  " PGCD = " , p )
```

Implantation en C#

Ecrivez le programme C# complet qui produise le dialogue suivant à la console (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

Proposition de squelette de classe C# à implanter :

```
class ApplicationEgyptien {
    static void Main(String[ ] args) {
       ……..
    }
    static int pgcd (int p, int q) {
       ……..
    }
 }
```

La méthode **pgcd** renvoie le pgcd des deux entiers p et q .

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
class ApplicationEgyptien {
 static void Main (string[ ] args) {
      System.Console.Write("Entrez le premier nombre : ");
     int p  = Int32.Parse( System.Console.ReadLine( ) ) ;
      System.Console.Write("Entrez le deuxième nombre : ");
     int q  = Int32.Parse( System.Console.ReadLine( ) ) ;
     if ( p*q != 0 )
       System.Console.WriteLine("Le pgcd de "+p+" et de "+q+" est "+pgcd(p,q));
     else
     System.Console.WriteLine("Le pgcd n'existe pas lorsque l'un des deux nombres est nul !");
  }

 static int pgcd (int p, int q) {
      while ( p != q)
      {
        if (p > q)  p  -= q;
        else q -= p;
       }
      return p;
  }

 }

}
```

Algorithme

Calcul de nombres premiers (boucles **while** et **do**…**while**)

Objectif : On souhaite écrire un programme C# de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même On opérera une implantation avec des boucles while et do...while.

Exemple : 31 est un nombre premier

Spécifications de l'algorithme :

```
Algorithme Premier
  Entrée: n  ∈ N
  Sortie: nbr   ∈ N
  Local: Est_premier  ∈ {Vrai , Faux}
         divis,compt  ∈N²;

    début
     lire(n);
     compt  ← 1;
     ecrire(2);
     nbr ← 3;
     Tantque(compt < n) Faire
      divis ← 3;
      Est_premier ← Vrai;
      Répéter
        Si reste(nbr par divis) = 0 Alors
         Est_premier ← Faux
        Sinon
         divis ← divis+2
        Fsi
      jusquà (divis > nbr / 2)ou (Est_premier=Faux);
      Si Est_premier =Vrai Alors
        ecrire(nbr);
        compt ← compt+1
      Fsi;
      nbr ← nbr+1
     Ftant
    FinPremier
```

Implantation en C#

Ecrivez le programme C# complet qui produise le dialogue suivant à la console (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Combien de nombres premiers : *5*
2
3
5
7
11

Classe C# solution

Avec une boucle while et une boucle do...while imbriquée :

On étudie la primalité de tous les nombres systématiquement

```
using System;
namespace CsExosAlgo1
{
 class ApplicationComptPremiers1 {
   static void Main(string[ ] args) {
       int divis, nbr, n, compt = 0 ;
       bool Est_premier;
       System.Console.Write("Combien de nombres premiers :  ");
       n = Int32.Parse( System.Console.ReadLine( ) ) ;
       System.Console.WriteLine( 2 );
       nbr = 3;
       while (compt < n-1)
       {
         divis = 2 ;
         Est_premier = true;
        do
        {
          if (nbr % divis == 0)   Est_premier=false;
          else divis = divis+1 ;
        }
        while ((divis <= nbr/2) && (Est_premier == true));
        if (Est_premier)
        {
           compt++;
           System.Console.WriteLine( nbr );
        }
         nbr++ ;
       }
    }
  }
}
```

La méthode **Main** affiche la liste des nombres premiers demandés.

Algorithme

Calcul de nombres premiers (boucles **for**)

Objectif : On souhaite écrire un programme C# de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même. On opérera une implantation avec des boucles for imbriquées.

Exemple : 23 est un nombre premier

Spécifications de l'algorithme : (*on étudie la primalité des nombres uniquement impairs)*

```
Algorithme Premier
  Entrée: n  ∈ N
  Sortie: nbr   ∈ N
  Local: Est_premier  ∈ {Vrai , Faux}
         divis,compt  ∈N²;

     début
      lire(n);
      compt  ← 1;
      ecrire(2);
      nbr ← 3;
      Tantque(compt < n) Faire
       divis ← 3;
       Est_premier ← Vrai;
       Répéter
         Si reste(nbr par divis) = 0 Alors
          Est_premier ← Faux
         Sinon
          divis ← divis+2
         Fsi
       jusquà (divis > nbr / 2)ou (Est_premier=Faux);
       Si Est_premier =Vrai Alors
         ecrire(nbr);
         compt ← compt+1
       Fsi;
       nbr ← nbr+2 // nbr impairs
      Ftant
     FinPremier
```

Implantation en C#

Ecrivez le programme C# complet qui produise le dialogue suivant à la console (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

```
Combien de nombres premiers :  5
  2
  3
  5
  7
  11
```

Classe C# solution

Avec deux boucles for imbriquées :

On étudie la primalité des nombres uniquement impairs

```
using System;
namespace CsExosAlgo1
{
 class ApplicationComptPremiers2 {
   static void Main(string[ ] args) {
       int divis, nbr, n, compt = 0 ;
       bool Est_premier;
       System.Console.Write("Combien de nombres premiers :  ");
       n = Int32.Parse( System.Console.ReadLine( ) ) ;
       System.Console.WriteLine( 2 );
       //-- primalité  uniquement des nombres impairs
       for( nbr = 3; compt < n-1; nbr += 2 )
       { Est_premier = true;
         for (divis = 2; divis<= nbr/2; divis++ )
          if ( nbr % divis == 0 )
          { Est_premier = false;
             break;
          }
         if (Est_premier)
         {
             compt++;
             System.Console.WriteLine( nbr );
         }
       }
    }
 }
}
```

La méthode **Main** affiche la liste des nombres premiers demandés.

Le fait de n'étudier la primalité que des nombres impairs accélère la vitesse d'exécution du programme, il est possible d'améliorer encore cette vitesse en ne cherchant que les diviseurs dont le carré est inférieur au nombre (test : **jusquà** ($divis^2$ > nbr)**ou** (Est_premier=Faux)).

Algorithme

Calcul du nombre d'or

Objectif : On souhaite écrire un programme C# qui calcule le nombre d'or utilisé par les anciens comme nombre idéal pour la sculpture et l'architecture. Si l'on considère deux suites numériques (U) et (V) telles que pour n strictement supérieur à 2 :

$$U_n = U_{n-1} + U_{n-2}$$

et

$$V_n = U_n / U_{n-1}$$

On montre que la suite (V) tend vers une limite appelée nombre d'or (nbr d'Or = 1,6180339887498948482045868343656 4).

Spécifications de l'algorithme :

n, U_n, U_{n1}, U_{n2} : sont des entiers naturels
V_n, V_{n1}, ε : sont des nombres réels

lire(ε); *// précision demandée*
  $U_{n2} \leftarrow 1$;
  $U_{n1} \leftarrow 2$;
  $V_{n1} \leftarrow 2$;
  $n \leftarrow 2$; *// rang du terme courant*
<u>Itération</u>
  $n \leftarrow n + 1$;
  $U_n \leftarrow U_{n1} + U_{n2}$;
  $V_n \leftarrow U_n / U_{n1}$;
  si $|V_n - V_{n1}| \leq \varepsilon$ **alors** Arrêt de la boucle ; *// la précision est atteinte*
  sinon
    $U_{n2} \leftarrow U_{n1}$;
    $U_{n1} \leftarrow U_n$;
    $V_{n1} \leftarrow V_n$;
  fsi
<u>fin Itération</u>
ecrire (V_n , n);

Ecrire un programme fondé sur la spécification précédente de l'algorithme du calcul du nombre d'or. Ce programme donnera une valeur approchée avec une précision fixée de ε du nombre d'or. Le programme indiquera en outre le rang du dernier terme de la suite correspondant.

Implantation en C#

On entre au clavier un nombre réel ci-dessous 0,00001, pour la précision choisie (ici 5 chiffres après la virgule), puis le programme calcule et affiche le Nombre d'or (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Précision du calcul ? : *0,00001*
Nombre d'Or = 1.6180328 // rang=14

Classe C# solution

Avec un boucle for :

```
using System;
namespace CsExosAlgo1
{
 class AppliNombredOr {
   static void Main(string[ ] args) {
       int n, Un, Un1=2, Un2=1 ;
       double Vn,Vn1=2, Eps  ;
       System.Console.Write("Précision du calcul ? :  ");
       //-- précision demandée (exemple 1e-4  ou 1e-2 ou 0,0001, …) :
       Eps = Double.Parse( System.Console.ReadLine( ) ) ;
       for (n=2; ; n++) //n est le rang du terme courant
       {
         Un = Un1 + Un2;
         Vn =(double)Un / (double)Un1;
        if (Math.Abs(Vn - Vn1) <= Eps)  break;
        else
        {
           Un2 = Un1;
           Un1 = Un;
           Vn1 = Vn;
        }
       }
       System.Console.WriteLine("Nombre d'Or = " + Vn+" // rang="+n);
   }
 }
}
```

Remarquons que nous proposons une boucle **for** ne contenant pas de condition de rebouclage dans son en-tête (donc en apparence infinie), puisque nous effectuerons le test "**si** |Vn - Vn1| <= Eps **alors** Arrêt de la boucle" qui permet l'arrêt de la boucle. Dans cette éventualité , la boucle **for** devra donc contenir dans son corps, une instruction de rupture de séquence.

Algorithme
Conjecture de Goldbach

Objectif : On souhaite écrire un programme C# afin de vérifier sur des exemples, la conjecture de GoldBach (1742), soit : "Tout nombre pair est décomposable en la somme de deux nombres premiers".

Dans cet exercice nous réutilisons un algorithme déjà traité (algorithme du test de la primalité d'un nombre entier), nous rappelons ci-après un algorithme indiquant si un entier "nbr" est premier ou non :

```
Algorithme Premier
   Entrée: nbr  ∈ N
   Local: Est_premier  ∈ {Vrai , Faux}
          divis,compt  ∈ N² ;

début
 lire(nbr);
  divis ← 3;
  Est_premier ← Vrai;
  Répéter
    Si reste(nbr par divis) = 0 Alors
     Est_premier ← Faux
    Sinon
     divis ← divis+2
    Fsi
  jusquà (divis > nbr / 2)ou (Est_premier=Faux);
  Si Est_premier = Vrai Alors
    ecrire(nbr est premier)
  Sinon
    ecrire(nbr n'est pas premier)
  Fsi
FinPremier
```

Spécifications de l'algorithme de Goldbach :

En deux étapes :

1. On entre un nombre pair n au clavier, puis on génère tous les couples (a,b) tels que a + b = n, en faisant varier a de 1 à n/2. Si l'on rencontre un couple tel que a et b soient simultanément premiers la conjecture est vérifiée.

2. On peut alors, au choix soit arrêter le programme, soit continuer la recherche sur un autre nombre pair.

Exemple :

Pour n = 10, on génère les couples :
(1,9), (2,8), (3,7), (4,6), (5,5)
on constate que la conjecture est vérifiée, et on écrit :
10 = 3 + 7
10 = 5 + 5

Conseils :

Il faudra traduire cet algorithme en fonction recevant comme paramètre d'entrée le nombre entier dont on teste la primalité, et renvoyant un booléen **true** ou **false** selon que le nombre entré a été trouvé ou non premier

On écrira la méthode booléenne **EstPremier** pour déterminer si un nombre est premier ou non, et la méthode **generCouples** qui génère les couples répondant à la conjecture.

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
 class ApplicationGoldBach {
  static void Main(string[ ] args) {
      int n;
      System.Console.WriteLine("Entrez un nombre pair (0 pour finir) :");
      while ( (n = Int32.Parse( System.Console.ReadLine( ) )) !=0 ){
      generCouples(n); }
  }
  static bool EstPremier(int m)  {
     int k ;
     for (k = 2 ; k <= m / 2 ; k++) {
         if (m % k == 0) {
         return  false;
         }
     }
     return true;
  }
  static void generCouples(int n) {
     if (n % 2 ==0) {
       for (int a = 1; a <= n/2; a++) {
         int  b;
            b = n - a;
         if ( EstPremier(a) && EstPremier(b) ) {
               System.Console.WriteLine(n+" = "+a+" + "+b);
         }
       }
    }
    else System.Console.WriteLine("Votre nombre n'est pas pair !");
  }
 }
}
```

Algorithme

Méthodes d'opérations sur 8 bits

Objectif : On souhaite écrire une application de manipulation interne des bits d'une variable entière non signée sur 8 bits. Le but de l'exercice est de construire une famille de méthodes de travail sur un ou plusieurs bits d'une mémoire. L'application à construire contiendra 9 méthodes :

Spécifications des méthodes :

1. Une méthode **BitSET** permettant de mettre à **1** un bit de rang fixé.
2. Une méthode **BitCLR** permettant de mettre à **0** un bit de rang fixé.
3. Une méthode **BitCHG** permettant de remplacer un bit de rang fixé par son complément.
4. Une méthode **SetValBit** permettant de modifier un bit de rang fixé.
5. Une méthode **DecalageD** permettant de décaler les bits d'un entier, sur la droite de **n** positions (introduction de **n** zéros à gauche).
6. Une méthode **DecalageG** permettant de décaler les bits d'un entier, sur la gauche de **n** positions (introduction de **n** zéros à droite).
7. Une méthode **BitRang** renvoyant le bit de rang fixé d'un entier.
8. Une méthode **ROL** permettant de décaler avec rotation, les bits d'un entier, sur la droite de **n** positions (réintroduction à gauche).
9. Une méthode **ROR** permettant de décaler avec rotation, les bits d'un entier, sur la gauche de **n** positions (réintroduction à droite).

Exemples de résultats attendus :

Prenons une variable X entier (par exemple sur 8 bits) X = **11100010**

1. **BitSET** (X,3) = X = **11101010**
2. **BitCLR** (X,6) = X = **10100010**
3. **BitCHG** (X,3) = X = **11101010**
4. **SetValBit**(X,3,1) = X = **11101010**
5. **DecalageD** (X,3) = X = **00011100**
6. **DecalageG** (X,3) = X = **00010000**
7. **BitRang** (X,3) = 0 ; BitRang (X,6) = 1 ...
8. **ROL** (X,3) = X = **00010111**
9. **ROR** (X,3) = X = **01011100**

Implantation en C#

La conception de ces méthodes ne dépendant pas du nombre de bits de la mémoire à opérer on choisira le type int comme base pour une mémoire. Ces méthodes sont classiquement des outils de manipulation de l'information au niveau du bit.

Lors des jeux de tests pour des raisons de simplicité de lecture il est conseillé de ne rentrer que des valeurs entières portant sur 8 bits.

Il est bien de se rappeler que le type primaire **int** est un type entier signé sur 32 bits (représentation en complément à deux).

Proposition de squelette de classe C# à implanter :

```
class Application8Bits {
    static void Main(string [ ] args){
        ……..      }
    static int BitSET (int nbr, int num) {  ……..      }
    static int BitCLR (int nbr, int num) {  ……..      }
    static int BitCHG (int nbr, int num) {  ……..      }
    static int SetValBit (int nbr, int rang, int val) {  ……..      }
    static int DecalageD (int nbr, int n) {  ……..      }
    static int DecalageG (int nbr, int n) {  ……..      }
    static int BitRang (int nbr, int rang) {  ……..   }
    static int ROL (int nbr, int n) {   ……..    }
    static int ROR (int nbr, int n) {  ……..    }
  }
```

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
class Application8Bits
{
//  Int32.MAX_VALUE : 32  bit = 2147483647
//  Int64.MAX_VALUE : 64 bits = 9223372036854775808

        static void Main(string[] args){
          int n,p,q,r,t;
          n =9;// 000...1001
          System.Console.WriteLine("n=9 : n="+toBinaryString(n));
          p = BitCLR(n,3);// p=1
          System.Console.WriteLine("BitCLR(n,3) ="+toBinaryString(p));
          q = BitSET(n,2);// q=13
          System.Console.WriteLine("BitSET(n,2) ="+toBinaryString(q));
          r = BitCHG(n,3);// r=1
```

```
 System.Console.WriteLine("BitCHG(n,3) ="+toBinaryString(r));
 t = BitCHG(n,2);// t=13
 System.Console.WriteLine("BitCHG(n,2) ="+toBinaryString(t));
 System.Console.WriteLine("p = "+p+", q = "+q+", r = "+r+", t = "+t);
 n =-2147483648;//1000.....00 entier minimal
 System.Console.WriteLine("n=-2^31 : n="+toBinaryString(n));
 p=ROL(n,3);// 000...000100 = p=4
 System.Console.WriteLine("p = "+p);
 n =-2147483648+1;//1000.....01 entier minimal+1
 System.Console.WriteLine("n=-2^31+1 : n="+toBinaryString(n));
 p=ROL(n,3);// 000...0001100 = p=12
 System.Console.WriteLine("p = "+p);
 n =3;//0000.....0 11
 System.Console.WriteLine("n=3 : n="+toBinaryString(n));
 p=ROR(n,1);//100000...001 = p=-2147483647
 System.Console.WriteLine("ROR(n,1) = "+p+"= "+toBinaryString(p));
 p=ROR(n,2);// 11000...000 = p= -1073741824
 System.Console.WriteLine("ROR(n,2) = "+p+"= "+toBinaryString(p));
 p=ROR(n,3);// 011000...000 = p= +1610612736 =2^30+2^29
 System.Console.WriteLine("ROR(n,3) = "+p+"= "+toBinaryString(p));
}
```

```
static string toBinaryString ( int n )
{ // renvoie l'écriture de l'entier n en représentation binaire
  string [ ] hexa = { "0000","0001","0010","0011","0100",
  "0101","0110","0111","1000","1001","1010",
  "1011","1100","1101","1110","1111" };
  string s = string.Format("{0:x}",n), res = "";
  for ( int i = 0; i < s.Length; i++ )
  {   char car=(char)s[i];
      if ((car <= '9')&&(car >= '0'))
        res = res+hexa[(int)car-(int)'0'];
      if ((car <= 'f')&&(car >= 'a'))
        res = res+hexa[(int)car-(int)'a'+10];
  }
  return res;
}

static  int  BitSET(int nbr, int num)
{ // positionne à 1 le bit de rang num
      int mask;
       mask =1<< num;
      return nbr | mask;
}

static  int  BitCLR(int nbr, int num)
{ // positionne à 0 le bit de rang num
      int mask;
       mask = ~ (1<< num);
      return nbr & mask;
}

static int  BitCHG(int nbr, int num)
{ // complémente le bit de rang num (0 si bit=1, 1 si bit=0)
      int mask;
       mask =1<< num;
      return nbr ^ mask;
}
```

```
static int DecalageD (int nbr, int n)
{ // décalage sans le signe de n bits vers la droite
      return nbr >> n ;
}

static int DecalageG (int nbr, int n)
{ // décalage de 2 bits vers la gauche
      return nbr << n ;
}

static int BitRang (int nbr, int rang)
{ //renvoie le bit de rang fixé
      return(nbr >> rang ) %2;
}

static int  SetValBit (int nbr, int rang,int val)
{ //positionne à val le bit de rang fixé
  return    val ==0 ? BitCLR( nbr , rang) : BitSET( nbr , rang) ;
}

static int ROL (int nbr, int n)
{ //décalage à gauche avec rotation
      int C;
      int N = nbr;
      for(int i=1; i<=n; i++)
       {
         C = BitRang(N,31);
         N = N <<1;
         N = SetValBit(N,0,C);
       }
      return N ;
}

static int ROR (int nbr,int n)
{ //décalage à droite avec rotation
      int C;
      int N = nbr;
      for(int i=1; i<=n; i++)
       {
         C = BitRang (N,0);
         N = N >> 1;
         N = SetValBit (N,31,C);
       }
      return N ;
}
```

} //--Application8Bits

}

String avec C#
phrase palindrome (deux versions)

Une phrase est dite palindrome si en éliminant les blancs entre les mots elle représente la même lecture dans les deux sens :

Exemple : elu par cette crapule ⇔ eluparc ettec rap ule

Voici le squelette du programme C# à écrire :

```
class palindrome
{
  static string compresser ( string s )
  {    …..

  }

  static string inverser ( string s )
  {    …..

  }
  static void Main ( string [ ] args )
  {
    System .Console.WriteLine ("Entrez une phrase :");
    string phrase = System .Console.ReadLine ( );
    string strMot = compresser ( phrase );
    string strInv = inverser ( strMot );
    if( strMot == strInv )
      System .Console.WriteLine ("phrase palindrome !");
   else
      System .Console.WriteLine ("phrase non palindrome !");
    System .Console.ReadLine ( );
  }
}
```

Travail à effectuer :

Ecrire les méthode **compresser** et **Inverser** , il est demandé d'écrire deux versions de la méthode **Inverser**.

- La première version de la méthode **Inverser** construira une chaîne locale à la méthode caractère par caractère avec une boucle **for** à un seul indice.
- La deuxième version de la méthode **Inverser** modifiera les positions des caractères ayant des positions symétriques dans la chaîne avec une boucle **for** à deux indices et en utilisant un tableau de **char**.

Classe C# solution

Le code de la méthode **compresser** :

```
static string compresser ( string s )
 {
   String strLoc = "";
  for( int i = 0 ; i < s.Length ; i ++ )
  {
   if( s[i] !=' ' && s[i] !=' , ' && s[i] !=' \ '' && s[i] !=' . ' )
       strLoc += s[i] ;
  }
  return strLoc ;
 }
```

La méthode **compresser** élimine les caractères non recevables comme : blanc, virgule, point et apostrophe de la **String** s passée en paramètre.

Remarquons que l'instruction strLoc +=s[i] permet de concaténer les caractères recevables de la chaîne locale strLoc, par balayage de la String s depuis le caractère de rang 0 jusqu'au caractère de rang s.Length-1.

La référence de String strLoc pointe à chaque tour de la boucle **for** vers un nouvel objet créé par l'opérateur de concaténation +

La première version de la méthode **Inverser** :

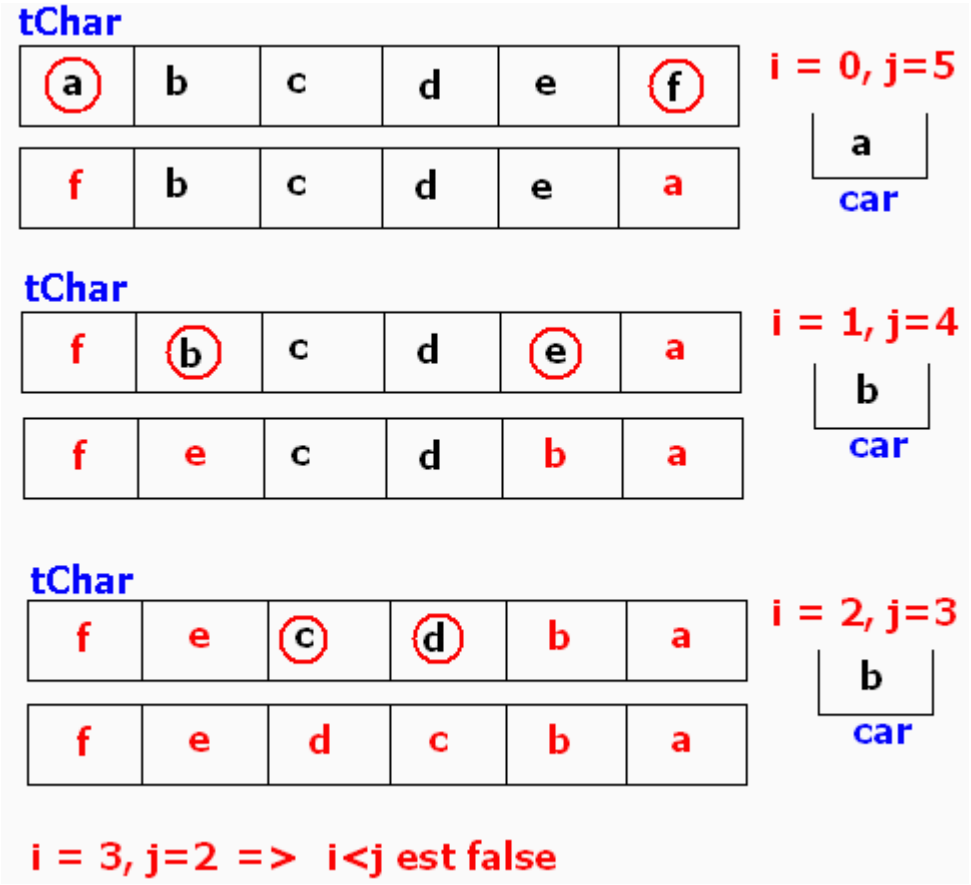
```
static string inverser ( string s )
{
  String strLoc = "";
  for( int i = 0 ; i < s.Length ; i ++ )
   strLoc = s[i] + strLoc ;
  return strLoc ;
}
```

La deuxième version de la méthode **Inverser** :

```
static string inverser ( string s )
{
  char [ ] tChar = s.ToCharArray ( );
  char car ;
  for( int i = 0 , j = tChar.Length - 1 ; i < j ; i ++ , j -- )
  {
     car = tChar[i] ;
     tChar[i] = tChar[j] ;
     tChar[j] = car ;
  }
  return new string ( tChar );
 }
```

```
for ( int  i = 0 , j = tChar.Length - 1 ;  i < j ;  i ++  , j -- )
```

Trace d'exécution de la boucle sur la chaîne s = "abcdef" :



String avec C#
convertir une date numérique en lettres

Objectif : Construire un programme permettant lorsqu'on lui fournit une date sous la forme numérique (3/2/5 où 3 indique le n° du jour de la semaine lundi=1, dimanche=7; le deuxième chiffre 2 indique le jour, enfin le troisième chiffre 5 indique le n° du mois) la convertit en clair (3/2/5 est converti en : mercredi 2 mai).

Proposition de squelette de classe C# à implanter :

```
class Dates
{
  enum LesJours  { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche };
  static string [ ] LesMois = { "janvier" , "février" , "mars" , "avril" ,"mai" , "juin" , "juillet" ,
                                "août" , "septembre" , "octobre" , "novembre" , "décembre" };

 static string mois ( int num ) { .... }
 static string jours ( int num ) { .... }
 static string numjour ( int num ) { .... }
 static void scanDate ( string date, out int jour, out int val, out int mois ) { .... }
 static string ConvertDate ( string date ) { .... }
 static void Main ( string [ ] args )
 {
   System .Console.WriteLine ( ConvertDate ("3/2/05"));
   System .Console.WriteLine ( ConvertDate ("5/7/1"));
   System .Console.WriteLine ( ConvertDate ("05/07/01"));
   System .Console.ReadLine ( );
 }
}
```

```
mercredi 2 mai
vendredi 7 janvier
vendredi 7 janvier
```

static string mois (int num)	Si le paramètre est un entier compris entre 1 et 12, elle renvoie le nom du mois (janvier,….,décembre), sinon elle renvoie ###.
static string jours (int num)	Si le paramètre est un entier compris entre 1 et 7, elle renvoie le nom du jour (lundi,….,dimanche), sinon elle renvoie ###.
static string numjour (int num)	Si le paramètre est un entier compris entre 1 et 31, elle le sous forme d'une string, sinon elle renvoie ###.
static void scanDate (string date, **out** int jour, **out** int val, **out** int mois)	Reçoit en entrée une date numérique formatée avec des séparateurs (/,.-) et ressort les 3 entiers la composant.
static string ConvertDate (string date)	Reçoit en entrée une date numérique formatée avec des sépaarteurs (/,.-) et renvoie sa conversion en forme littérale.

Classe C# solution

```
class Dates
{

  enum LesJours  { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche };
  static string [ ] LesMois = { "janvier" , "février" , "mars" , "avril" ,"mai" , "juin" , "juillet" ,
                                "août" , "septembre" , "octobre" , "novembre" , "décembre" };

  static string mois ( int num )
  {
   if( num >= 0 && num < 12 )
    return LesMois[num] ;
   else
    return "mois ####";
  }
  static string jours ( int num )
  {
    string [ ] tablej = Enum.GetNames ( typeof ( LesJours ));
   if( num >= 0 && num < 7 )
    return tablej [num] ;
   else
    return "jour ####";
  }
  static string numjour ( int num )
  {
   if( num >= 1 && num <= 31 )
    return num.ToString ( ) ;
   else
    return "n° ####";
  }
  static void scanDate ( string date, out int jour, out int val, out int mois )
  {
    string delimStr = "/.:-";
    char [ ] delimiteurs = delimStr.ToCharArray ( );
    string [ ] strInfos = date.Split ( delimiteurs , 3 );
    jour = Convert.ToInt32 ( strInfos[0] ) - 1 ;
    val = Convert.ToInt32 ( strInfos[1] );
    mois = Convert.ToInt32 ( strInfos[2] ) - 1 ;
  }
  static string ConvertDate ( string date )
  {
    int lejour, laval, lemois ;
    scanDate ( date, out lejour, out laval, out lemois );
    return jours ( lejour ) + " " + numjour ( laval ) + " " + mois ( lemois );
  }
  static void Main ( string [ ] args )
  {
    System .Console.WriteLine ( ConvertDate ("3-2-05"));
    System .Console.WriteLine ( ConvertDate ("5/7/1"));
    System .Console.WriteLine ( ConvertDate ("05.07.01"));
    System .Console.ReadLine ( );
  }
}
```

Tous les noms du type enum sont automatiquement stockés dans le tableau de string tablej .

La méthode Split de la classe **string** est très utile ici, elle constitue un scanner de chaîne : elle découpe automatiquement la chaîne en 3 éléments dans un tableau de string.

```
mercredi 2 mai
vendredi 7 janvier
vendredi 7 janvier
```

String avec C#
convertir un nombre écrit en chiffres Romains

Objectif : Construire un programme permettant de convertir un nombre romain écrit avec les lettres M,D,C,L,X,V,I en un entier décimal.

Classe C# solutions (récursive et itérative)

```
class Romains
{
  static int tradRom ( char car )
  {
    string carRomainStr = "MDCLXVI";
   if( carRomainStr.IndexOf ( car ) !=- 1 )
    switch( car )
   {
    case 'M' :return 1000 ;
    case 'D' :return 500 ;
    case 'C' :return 100 ;
    case 'L' :return 50 ;
    case 'X' :return 10 ;
    case 'V' :return 5 ;
    case 'I' :return 1 ;
    default : return 0 ;
   }
   else
    return 0 ;
  }
```

Méthode récursive	Méthode iétrative

`static int recurRom ( string chiff, int i )`
`{`
`int valLettre = tradRom ( chiff[i] );`
`if( i == chiff.Length - 1 )`
`return valLettre;`
`else`
`if (valLettre < tradRom ( chiff[i + 1] ))`
`return recurRom ( chiff, ++ i ) - valLettre;`
`else`
`return recurRom ( chiff, ++ i ) + valLettre;`
`}`
	`static int iterRom ( string chiff ) {`
	`int som = 0, valLettre = 0 ;`
	`for( int i = 0 ; i < chiff.Length ; i++ ) {`
	`valLettre = tradRom ( chiff[i] );`
	`if( i == chiff.Length - 1 )`
	`break;`
	`if (valLettre < tradRom ( chiff[i + 1] ))`
	`som = som - valLettre;`
	`else`
	`som = som + valLettre;`
	`}`
	`return som + valLettre;`
	`}`

```
  static void Main ( string [ ] args ) {
    System .Console.WriteLine ( recurRom ("MCDXCIV" ,0 ));
    System .Console.WriteLine ( iterRom ("MCDXCIV"));
    System .Console.ReadLine ();
  }
}
```

Algorithme

Tri à bulles sur un tableau d'entiers

Objectif : Ecrire un programme C# implémentant l'algorithme du tri à bulles.

Spécifications de l'algorithme :

```
Algorithme Tri_a_Bulles
 local:   i , j , n, temp ∈  Entiers naturels
 Entrée - Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
début
 pour i de n jusquà 1 faire // recommence une sous-suite (a1, a2, ... , ai)
  pour j de 2 jusquà i faire // échange des couples non classés de la sous-suite
   si Tab[ j-1 ] > Tab[ j ] alors // aj-1et aj non ordonnés
     temp ←Tab[ j-1 ] ;
     Tab[ j-1 ] ← Tab[ j ] ;
     Tab[ j ]  ← temp //on échange les positions de aj-1et aj
   Fsi
  fpour
 fpour
Fin Tri_a_Bulles
```

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
class ApplicationTriBulle  {
 static int[ ] table ; // le tableau à trier, par exemple 19 éléments de l'index 1 à l'index 19
static void AfficherTable ( ) {
    // Affichage du tableau
       int n = table.Length-1;
       for ( int i = 1; i <= n; i++)
          System.Console.Write (table[i]+" , ");
        System.Console.WriteLine ( );
    }

    static void InitTable  ( ) {
       int[ ] tableau = { 0 ,25, 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 ,
                          89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52 };
       table = tableau;
    }

 static void Main(string[ ] args) {
        InitTable ( );
        System.Console.WriteLine ("Tableau initial :");
        AfficherTable ( );
        TriBulle ( );
        System.Console.WriteLine ("Tableau une fois trié :");
```

```
        AfficherTable ( );
        System.Console.Read();
    }
  static void TriBulle ( ) {
    // sous-programme de Tri à bulle  : on trie les éléments du n°1 au  n°19
        int n = table.Length-1;
        for ( int i = n; i>=1; i--)
        for ( int j = 2; j <= i; j++)
          if (table[j-1] > table[j]){
            int temp = table[j-1];
            table[j-1] = table[j];
            table[j] = temp;
          }
  /* Dans le cas où l'on démarre le tableau à l'indice zéro
     on change les bornes des indices i et j:
        for ( int i = n; i >= 0; i--)
        for ( int j = 1; j <= i; j++)
           if ....... reste identique
  */
    }

   }
  }
```

Tableau initial (n°1 au n°19):
25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52

Tableau une fois trié (n°1 au n°19) :
7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Autre version depuis l'indice zéro :

Tableau initial (n°0 au n°19):
0, 25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52

Tableau une fois trié (n°0 au n°19) :
0, 7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Algorithme

Tri par insertion sur un tableau d'entiers

Objectif : Ecrire un programme C# implémentant l'algorithme du tri par insertion.

Spécifications de l'algorithme :

```
Algorithme Tri_Insertion
 local:   i , j , n, v ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments (le même tableau)


{ dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle tantque .. faire si
l'indice j n'est pas inférieur à 1, elle aura une valeur inférieure à toute valeur possible de la liste
}


début


 pour i de2 jusquà n faire// la partie non encore triée (ai, ai+1, ... , an)
    v ← Tab[ i ] ;  // l'élément frontière : ai
    j ← i ;        // le rang de l'élément frontière
     Tantque Tab[ j-1 ]> v faire//on travaille sur la partie déjà triée (a1, a2, ... , ai)
            Tab[ j ]  ←Tab[ j-1 ]; // on décale l'élément
             j ← j-1;              // on passe au rang précédent
     FinTant ;
     Tab[ j ]  ← v //on recopie ai dans la place libérée
  fpour


Fin Tri_Insertion
```

Classe C# solution

On utilise une sentinelle placée dans la cellule de rang 0 du tableau, comme le type d'élément du tableau est un **int**, nous prenons comme valeur de la sentinelle une valeur négative très grande par rapport aux valeurs des éléments du tableau; par exemple le plus petit élément du type int, soit la valeur Integer.MIN_VALUE.

```
using System;
namespace CsExosAlgo1
{
class ApplicationTriInsert {
 static int[] table ; // le tableau à trier, par exemple 19 éléments
  /* Tri avec sentinelle :
   * dans la cellule de rang 0 se trouve une sentinelle (Int32.MinValue)
   * chargée d'éviter de tester dans la boucle tantque .. faire
   * si l'indice j n'est pas inférieur à 1, elle aura une valeur
   * inférieure à toute valeur possible de la liste.
   */
 static void AfficherTable ( ) {
     // Affichage du tableau
        int n = table.Length-1;
        for ( int i = 1; i <= n; i++)
           System.Console.Write (table[i]+" , ");
         System.Console.WriteLine ( );
    }

     static void InitTable  ( )
    { // sentinelle dans la cellule de rang zéro
        int[ ] tableau = { Int32.MinValue ,25, 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 ,
                           89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52 };
        table = tableau;
    }

  static void Main(string[ ] args) {
         InitTable ( );
         System.Console.WriteLine ("Tableau initial :");
         AfficherTable ( );
         TriInsert  ( );
         System.Console.WriteLine ("Tableau une fois trié :");
        AfficherTable ( );
        System.Console.Read();
   }
 static void TriInsert  ( ) {
  // sous-programme de Tri par insertion : on trie les éléments du n°1 au  n°19
        int n = table.Length-1;
        for ( int i = 2; i <= n; i++)
        { // la partie non encore triée (ai, ai+1, ... , an)
         int v = table[i];            // l'élément frontière : ai
         int j = i;                   // le rang de l'élément frontière
         while (table[ j-1 ] > v)
        { //on travaille sur la partie déjà triée (a1, a2, ... , ai)
           table[ j ] = table[ j-1 ]; // on décale l'élément
              j = j-1;                // on passe au rang précédent
         }
         table[ j ]  = v ;            //on recopie ai dans la place libérée
        }
   }
  }
 }
```

Tableau initial :
25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52

Tableau une fois trié :
7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Algorithme

Recherche linéaire dans une table non triée

Objectif : Ecrire un programme C# effectuant une recherche séquentielle dans un tableau linéaire (une dimension) non trié

TABLEAU NON TRIE

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments non rangés.

- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

Version **Tantque** avec "**et alors**" (opérateur et optimisé)

```
 i ← 1 ;
Tantque (i ≤ n) et alors (t[i] ≠ Elt) faire
      i ← i+1
finTant;
si i ≤ n alors rang ← i
sinon rang ← -1
Fsi
```

Version **Tantque** avec "**et**" (opérateur et non optimisé)

```
 i ← 1 ;
Tantque (i < n) et (t[i] ≠ Elt) faire
      i ← i+1
finTant;
si t[i] = Elt alors rang ← i
sinon rang ← -1
Fsi
```

Version **Tantque** avec sentinelle en fin de tableau (rajout d'une cellule)

```
t[n+1] ← Elt ; // sentinelle rajoutée
```

```
i ← 1 ;
Tantque (i ≤ n) et alors (t[i] ≠ Elt) faire
      i ←  i+1
finTant;
si  i ≤ n  alors rang ←  i
sinon rang ← -1
Fsi
```

Version **Pour** avec instruction de sortie (**Sortirsi**)

```
pour i ← 1 jusquà n faire
  Sortirsi t[i] = Elt
fpour;
si  i ≤ n  alors rang ← i
sinon rang ←  -1
Fsi
```

Traduire chacune des quatre versions sous forme d'une méthode C#.

Proposition de squelette de classe C# à implanter :

```
class ApplicationRechLin  {

  static void AfficherTable ( int[] table )     {
          // Affichage du tableau
   }

   static void InitTable  ( )     {
              // remplissage du tableau      }

  static int RechSeq1( int[] t, int Elt )      {
      // Version Tantque avec "et alors" (opérateur et optimisé)       }

 static int RechSeq2( int[] t, int Elt )    {
      // Version Tantque avec "et" (opérateur non optimisé)    }

  static int RechSeq3( int[] t, int Elt )     {
     // Version Tantque avec sentinelle en fin de tableau     }

 static int RechSeq4( int[] t, int Elt )    {
    // Version Pour avec instruction de sortie break   }

 static void Main(string[ ] args)
 {
       InitTable ( );
       System.Console.WriteLine("Tableau initial :");
       AfficherTable (table );
       int x = Int32.Parse(System.Console.ReadLine( )),  rang;
       //rang = RechSeq1( table, x );
       //rang = RechSeq2( table, x );
       //rang = RechSeq3( tableSent, x );
       rang = RechSeq4( table, x );
       if (rang > 0)
         System.Console.WriteLine("Elément "+x+" trouvé en : "+rang);
```

```
        else  System.Console.WriteLine("Elément "+x+" non trouvé !");
          System.Console.Read();
    }
  }
}
```

Classe C# solution

Les différents sous-programmes C# implantant les 4 versions d'algorithme de recherche linéaire (*table non triée*) :

static int RechSeq1(**int**[] t, **int** Elt)
{
  int i = 1; **int** n = t.Length-1;
  while ((i <= n) **&&** (t[i] != Elt)) i++;
  if (i<=n)
  return i;
  else
  return -1;
}
	static int RechSeq2(**int**[] t, **int** Elt)
	{
	  int i = 1; **int** n = t.Length-1;
	  while ((i < n) **&** (t[i] != Elt)) i++;
	  if (t[i] == Elt)
	  return i;
	  else
	  return -1;
	}

static int RechSeq3(**int**[] t, **int** Elt)
{
  int i = 1; **int** n = t.Length-**2**;
  t[n+1]= Elt ; *//sentinelle*
  while ((i <= n) **&** (t[i] != Elt)) i++;
  if (i<=n)
  return i;
  else
  return -1;
}
	static int RechSeq4(**int**[] t, **int** Elt)
	{
	  int i = 1; **int** n = t.Length-1;
	  for(i = 1; i <= n ; i++)
	  if (t[i] == Elt)
	  break;
	  if (i<=n)
	  return i;
	  else
	  return -1;
	}

Une classe complète permettant l'exécution des sous-programmes précédents :

```
using System;
namespace CsExosAlgo1
{
  class ApplicationRechLin {

        static int max = 20;
        static int[ ] table; // cellules à examiner de 1 à 19
        static int[ ] tableSent = new int[max+1] ; // le tableau à examiner de 1 à 20

        static void AfficherTable (int[ ] t ) {
            // Affichage du tableau
          int n = t.Length-1;
          for ( int i = 1; i <= n; i++)
            System.Console.Write(t[i]+" , ");
           System.Console.WriteLine( );
        }
         static void InitTable  ( ) {
            // remplissage du tableau
         int[ ] tableau = { 14, 35, 84, 49, 5 , 94, 89, 58, 61, 4 , 39, 58 , 57 , 99 , 12 , 24 , 9 , 81 , 80 };
          table = tableau;
```

```
    int n = table.Length-1;
    for ( int i = 1; i <= table.Length-1; i++)
          tableSent[i] = table[i];
  }
  static int RechSeq1( int[] t, int Elt ) {
   // Version Tantque avec "et alors" (opérateur et optimisé)
    int i = 1; int n = t.Length-1;
    while ((i <= n) && (t[i] != Elt)) i++;
    if (i<=n)
      return i;
    else
       return -1;
  }
  static int RechSeq2( int[] t, int Elt ) {
   // Version Tantque avec "et" (opérateur non optimisé)
    int i = 1; int n = t.Length-1;
    while ((i < n) & (t[i] != Elt)) i++;
    if (t[i] == Elt)
      return i;
    else
       return -1;
  }
  static int RechSeq3( int[] t, int Elt ) {
   // Version Tantque avec sentinelle en fin de tableau
    int i = 1; int n = t.Length-2;
    t[n+1]= Elt ;  // sentinelle rajoutée
    while ((i <= n) & (t[i] != Elt)) i++;
    if (i<=n)
      return i;
    else
       return -1;
  }
  static int RechSeq4( int[] t, int Elt ) {
   // Version Pour avec instruction de sortie break
    int i = 1; int n = t.Length-1;
    for(i = 1; i <= n  ; i++)
      if (t[i] == Elt)
         break;
    if (i<=n)
      return i;
    else
       return -1;
  }
  static void Main(String[ ] args) {
    InitTable ( );
    System.Console.WriteLine("Tableau initial :");
    AfficherTable (table );
    int x = Int32.Parse(System.Console.ReadLine( )),  rang;
    //rang = RechSeq1( table, x );
    //rang = RechSeq2( table, x );
    //rang = RechSeq3( tableSent, x );
    rang = RechSeq4( table, x );
    if (rang > 0)
       System.Console.WriteLine("Elément "+x+" trouvé en : "+rang);
    else  System.Console.WriteLine("Elément "+x+" non trouvé !");
    System.Console.Read();
  }
} }
```

Algorithme

Recherche linéaire dans table déjà triée

Objectif : Ecrire un programme C# effectuant une recherche séquentielle dans un tableau linéaire (une dimension) trié avant recherche.

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.

- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

On peut reprendre sans changement les algorithmes précédents travaillant sur un tableau non trié.

On peut aussi utiliser le fait que le dernier élément du tableau est **le plus grand élément** et s'en servir comme une sorte de **sentinelle**. Ci-dessous deux versions utilisant cette dernière remarque.

Version **Tantque** :

```
si t[n] ≥ Elt alors
  i ← 1 ;
 Tantque t[i] < Elt faire
    i ← i+1;
  finTant;
   si t[i] = Elt alors
     Renvoyer rang ← i // retour du résultat et sortie du programme
   Fsi
Fsi;
Renvoyer rang ← -1 // retour du résultat et sortie du programme
```

Version **pour** :

```
si t[n] ≥ Elt alors rang ← -1
   pour i ← 1 jusquà n-1 faire
     Sortirsi t[i] ≥ Elt //sortie de boucle
   fpour;
   si t[i] = Elt alors
      Renvoyer rang ← i // retour du résultat et sortie du programme
   Fsi
Fsi;
Renvoyer rang ← -1 // retour du résultat et sortie du programme
```

Ecrire chacune des méthodes associées à ces algorithmes (prendre soin d'avoir trié le tableau auparavant par exemple par une methode de tri), squelette de classe proposé :

Classe C# solution

Les deux méthodes C# implantant les 2 versions d'algorithme de recherche linéaire (*table triée*) :

static int RechSeqTri1(**int**[] t, **int** Elt)
{
int i = 1; int n = t.Length-1;
if (t[n] >= Elt) {
while (t[i] < Elt) i++;
if (t[i] == Elt)
return i ;
}
return -1;
}
	static int RechSeqTri2(**int**[] t, **int** Elt)
	{
	int i = 1; **int** n = t.Length-1;
	if (t[n] >= Elt) {
	for (i = 1; i <= n; i++)
	if (t[i] == Elt)
	return i;
	}
	return -1;
	}

Une classe complète utilisant ces deux méthodes :

```
using System;
namespace CsExosAlgo1
{
class ApplicationRechLinTrie  {

    static int[ ] table  ; //cellules à examiner de 1 à 19



    static void AfficherTable (int[] t ) {
        // Affichage du tableau
      int n = t.Length-1;
      for ( int i = 1; i <= n; i++)
        System.Console.Write(t[i]+" , ");
       System.Console.WriteLine( );
    }
     static void InitTable  ( ) {
        // remplissage du tableau la cellule de rang 0 est inutilisée
      int[ ] tableau = { 0, 14 , 35 , 84 , 49 , 50 , 94 , 89 , 58 , 61 , 47 ,
                         39 , 58 , 57 , 99 , 12 , 24 , 9 , 81 , 80 };
      table = tableau;
    }
     static void TriInsert ( ) {
      // sous-programme de Tri par insertion :
      int n = table.Length-1;
      for ( int i = 2; i <= n; i++) {
        int v = table[i];
        int j = i;
        while (table[ j-1 ] > v) {
           table[ j ] = table[ j-1 ];
           j = j-1;
         }
         table[ j ]  = v ;
```

```
            }
          }

    static int RechSeqTri1( int[] t, int Elt ) {
      int i = 1; int n = t.Length-1;
      if (t[n] >= Elt) {
        while (t[i] < Elt) i++;
          if (t[i] == Elt)
              return  i ;
      }
      return  -1;
    }

  static int RechSeqTri2( int[] t, int Elt ) {
      int i = 1; int n = t.Length-1;
      if (t[n] >= Elt) {
        for (i = 1; i <= n; i++)
            if (t[i] == Elt)
              return  i;
      }
      return  -1;
  }
      static void Main(string[ ] args) {
        InitTable ( );
        System.Console.WriteLine("Tableau initial :");
        AfficherTable (table );
        TriInsert ( );
        System.Console.WriteLine("Tableau trié :");
        AfficherTable (table );
        int x = Int32.Parse(System.Console.ReadLine( )), rang;
        //rang = RechSeqTri1( table, x );
        rang = RechSeqTri2( table, x );
         if (rang > 0)
           System.Console.WriteLine("Elément "+x+" trouvé en : "+rang);
        else  System.Console.WriteLine("Elément "+x+" non trouvé !");
        System.Console.Read();
      }
  }
}
```

Algorithme

Recherche dichotomique dans une table

Objectif : effectuer une recherche dichotomique dans un tableau linéaire déjà trié.

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments **triés par ordre croissant**.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

Au lieu de rechercher séquentiellement du premier jusqu'au dernier, on compare l'élément **Elt** à chercher au contenu du milieu du tableau. Si c'est le même, on retourne le rang du milieu, sinon l'on recommence sur la première moitié (ou la deuxième) si l'élément recherché est plus petit (ou plus grand) que le contenu du milieu du tableau.

Version itérative

```
bas, milieu, haut, rang : entiers

bas ← 1;
haut ← N;
Rang ← -1;
repéter
  milieu ←  (bas + haut) div 2;
  si x = t[milieu] alors
     Rang ← milieu
  sinon si t[milieu] < x alors
            bas ←  milieu + 1
         sinon
            haut ←  milieu-1
         fsi
  fsi
jusquà ( x = t[milieu] ) ou ( bas haut )
```

Implanter une méthode en C# que vous nommerez RechDichoIter qui recevant en entrée un tableau et un élément à chercher, renverra le rang de l'élément selon les spécifications ci-haut. N'oubliez pas de trier le tableau avant d'invoquer la méthode de recherche. Ci-dessous une suggestion de rédaction de la méthode **Main** :

```
static void Main(string[ ] args)
  {
      InitTable ( );
      System.Console.WriteLine("Tableau initial :");
      AfficherTable ( );
```

```
        TriInsert ( );
        System.Console.WriteLine("Tableau trié :");
        AfficherTable ( );
         int x = Int32.Parse(System.Console.ReadLine( )), rang;
        rang = RechDichoIter( table, x );
         if (rang 0)
           System.Console.WriteLine("Elément "+x+" trouvé en : "+rang);
        else  System.Console.WriteLine("Elément "+x+" non trouvé !");
        System.Console.Read();
    }
```

Proposition de squelette de classe C# à implanter :

```
class  ApplicationRechDicho
{

        static void AfficherTable ( )     {
         // Affichage du tableau    ............          }

        static void InitTable  ( )   {
          ............          }

        static void TriInsert ( )    {
            // sous-programme de Tri par insertion
             ............          }

        static int RechDichoIter( int[ ] t, int Elt )          {
            // Recherche par dichotomie
            ............
        }

        static void Main(string[ ] args)          {
            ............          }
    }
```

Classe C# solution

```
using System;
namespace CsExosAlgo1
{
class  ApplicationRechDicho
{

     static int[ ] table; // le tableau à examiner cellules de 1 à 19
    static void AfficherTable ( )
    {
     // Affichage du tableau
        int n = table.Length-1;
        for ( int i = 1; i <= n; i++)
          System.Console.Write (table[i]+" , ");
        System.Console.WriteLine( );
     }
    static void InitTable  ( )
    {
     // La cellule de rang zéro est inutilisée (examen sur 19 éléments)
        int[ ] tableau = { 0 , 53 , 77 , 11 , 72 , 28 , 43 , 65 , 83 , 39 , 73 ,
```

```
                                82 , 69 , 65 , 4 , 95 , 46 , 12 , 87 , 75 };
        table = tableau;
    }
    static void TriInsert ( )
    {
        // sous-programme de Tri par insertion :
        int n = table.Length-1;
        for ( int i = 2; i <= n; i++) {
          int v = table[i];
          int j = i;
          while (table[ j-1 ] > v) {
              table[ j ] = table[ j-1 ];
              j = j-1;
            }
            table[ j ]  = v ;
        }
    }
    static int RechDichoIter( int[ ] t, int Elt )
    {
        int n = t.Length-1;
        int bas =  1, haut =  n, milieu ;
        int  Rang =  -1;
        do{
           milieu =  (bas + haut) / 2;
          if ( Elt == t[milieu])  Rang =  milieu ;
          else if  ( t[milieu] < Elt )  bas =  milieu + 1 ;
          else haut =  milieu-1 ;
        }
        while ( ( Elt != t[milieu] ) & ( bas <= haut ) );
        return Rang;
    }
    static void Main(string[ ] args)
    {
        InitTable ( );
        System.Console.WriteLine("Tableau initial :");
        AfficherTable ( );
        TriInsert ( );
        System.Console.WriteLine("Tableau trié :");
        AfficherTable ( );
         int x = Int32.Parse(System.Console.ReadLine( )), rang;
        rang = RechDichoIter( table, x );
         if (rang > 0)
           System.Console.WriteLine("Elément "+x+" trouvé en : "+rang);
        else  System.Console.WriteLine("Elément "+x+" non trouvé !");
        System.Console.Read();
      }
    }
  }
```

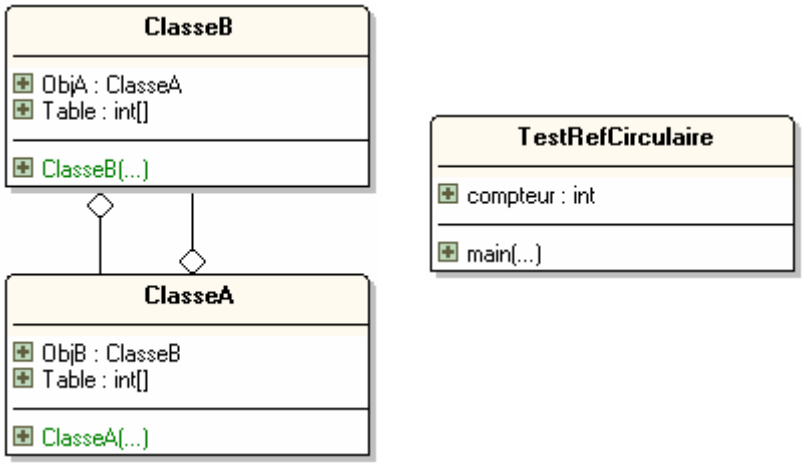
Remarque : Ces exercices sont purement académiques et servent à apprendre à utiliser le langage sur des algorithmes classiques, car la classe **Array** contient déjà une méthode static de recherche dichotomique dans un tableau ***t*** trié au préalable par la méthode static Sort de la même classe **Array** :

> **public static** int **BinarySearch (** Array ***t*** **,** object ***elt*** **);**
>
> **public static** void **Sort(** Array ***t*** **);**

Classes, objet et IHM

Problème de la référence circulaire

On donne trois classes ClasseA, ClasseB, TestRefCirculaire :



- La classe ClasseA possède une référence ObjB à un objet public de classe ClasseB, possède un attribut Table qui est un tableau de 50 000 entiers, lorsqu'un objet de ClasseA est construit il incrémente de un le champ static compteur de la classe TestRefCirculaire et instancie la référence ObjB.

- La classe ClasseB possède une référence ObjA à un objet public de classe ClasseA, possède un attribut Table qui est un tableau de 50 000 entiers, lorsqu'un objet de ClasseB est construit il incrémente de un le champ static compteur de la classe TestRefCirculaire et instancie la référence ObjA.

- La classe TestRefCirculaire ne possède qu'un attribut de classe : le champ public entier compteur initialisé à zéro au départ, et la méthode principale Main de lancement de l'application.

Implémentez ces trois classes en ne mettant dans le corps de la méthode Main qu'une seule instruction consistant à instancier un objet local de classe ClasseA, puis exécuter le programme et expliquez les résultats obtenus.

Programme C# solution

Code source demandé :

```
using System;

namespace ConsoleGarbageRefCirc {

 class ClasseA  {
```

```
 public ClasseB ObjB;
 public int[ ]Table = new int[50000];

 public ClasseA( )  {
  TestRefCirculaire.compteur++;
  System.Console.WriteLine("Création objet ClasseA n° "+TestRefCirculaire.compteur);
  ObjB = new ClasseB( );
 }
}

class ClasseB  {
 public ClasseA ObjA;
 public int[ ]Table = new int[50000];

 public ClasseB( )  {
  TestRefCirculaire.compteur++;
  System.Console.WriteLine("Création objet ClasseB n° "+TestRefCirculaire.compteur);
  ObjA = new ClasseA( );
 }
}

class TestRefCirculaire {
 public static int compteur=0;
 [STAThread]
 static void Main(string[ ] args)  {
  ClasseA ObjA = new ClasseA( );
 }
}
}
```

Résultats d'exécution :

```
Création objet ClasseA n° 1
Création objet ClasseB n° 2
Création objet ClasseA n° 3
Création objet ClasseB n° 4
Création objet ClasseA n° 5
Création objet ClasseB n° 6
Création objet ClasseA n° 7
Création objet ClasseB n° 8
Création objet ClasseA n° 9
Création objet ClasseB n° 10
      :
      :
      :
Création objet ClasseA n° 7053
Création objet ClasseB n° 7054
Création objet ClasseA n° 7055
```

Le système envoie une notification d'exception OutOfMemoryException, puis arrête le programme :

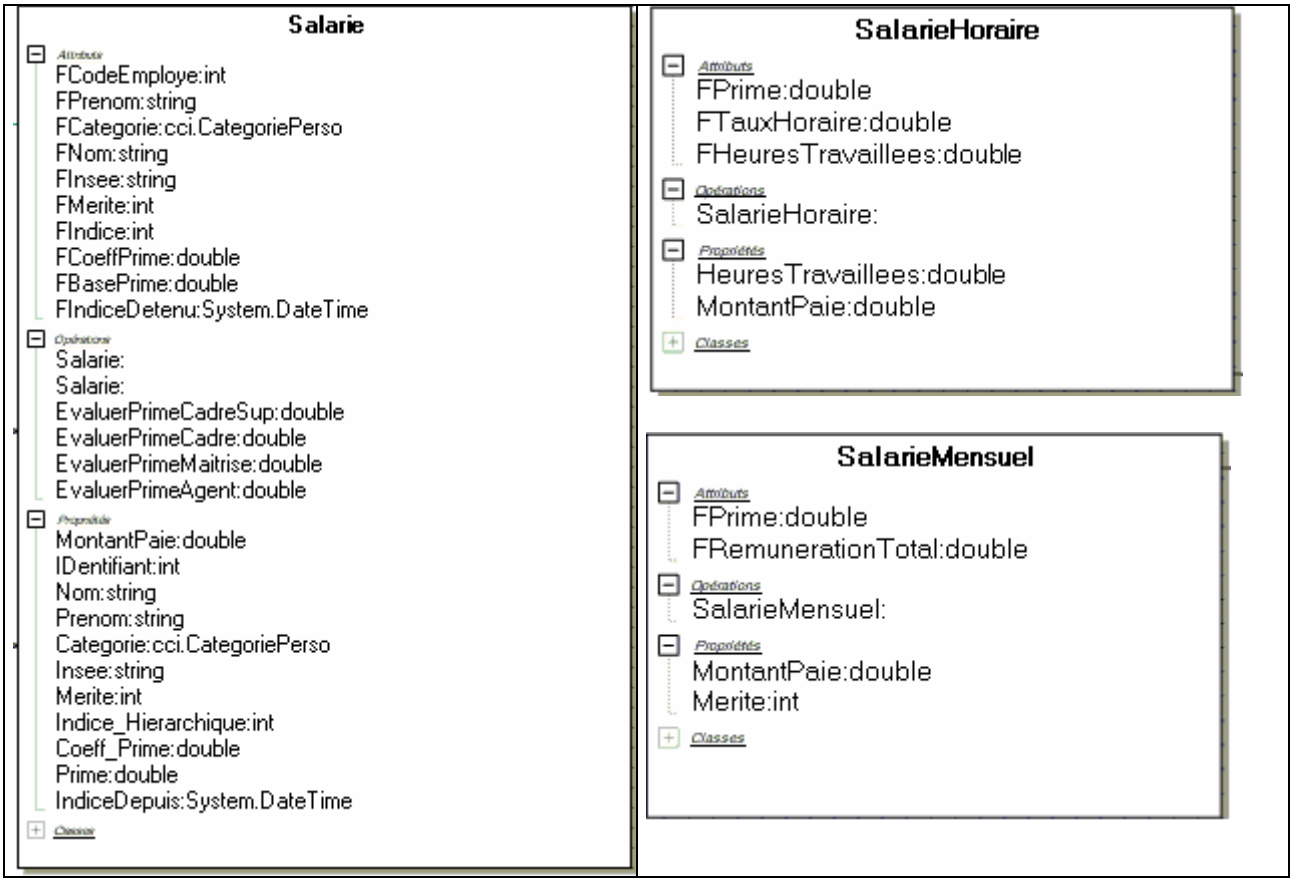
Explications :

L' instanciation d'un objetA provoque l'instanciation d'un objetB qui lui même provoque l'instanciation d'un autre objetA qui à son tour instancie un objetB etc… A chaque instanciation d'un objet de ClasseA ou de ClasseB, un tableau de 50 000 entiers est réservé dans la pile d'exécution, nous voyons sur l'exemple que la 7055ème instanciation a été fatale car la pile a été saturée ! L'instanciation d'un seul objet a provoqué la saturation de la mémoire car les ClasseA et ClasseB sont liées par une association de double référence ou référence circulaire, il faut donc faire attention à ce genre de configuration.

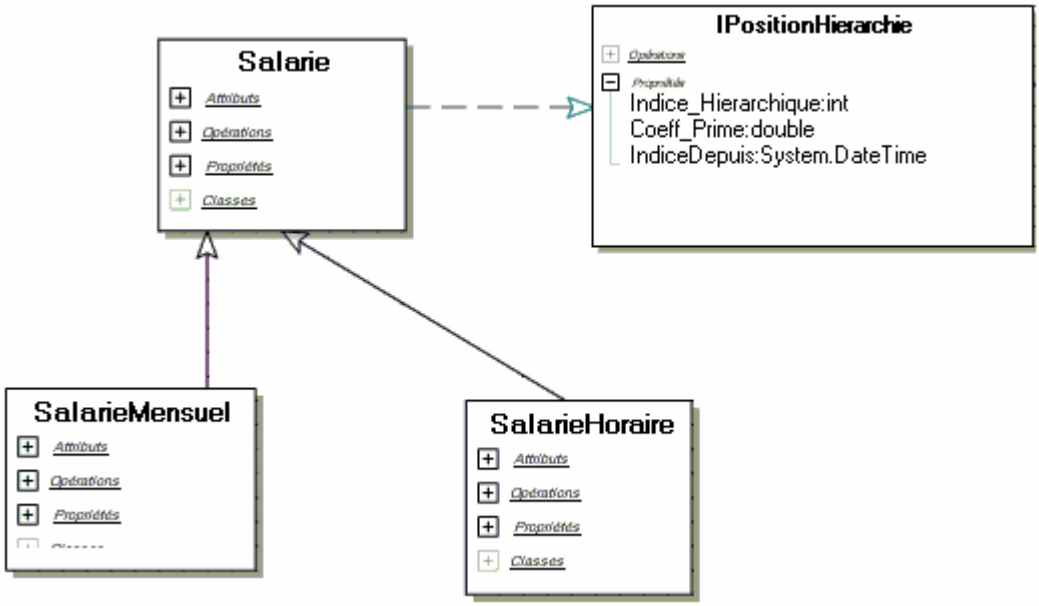
Classes, objet et IHM

Classe de salariés dans une entreprise fictive

Soient les diagrammes de classes suivants censés modèliser le type de salarié employé dans une entreprise. Nous distinguons deux genres de salariés ceux qui sont mensualisés et ceux qui sont payés à l'heure :



La classe **Salarie** implémente l'interface **IpositionHierarchie** et les classes **SalarieMensuel** et **SalarieHoraire** héritent toutes deux de la classe Salarie :



On propose une implémentation partielle des squelettes de ces classes:

```
using System ;
using System.Collections ;
using System.Threading ;

namespace cci
{
 enum CategoriePerso { Cadre_Sup,Cadre,Maitrise,Agent,Autre }
```

```
/// <summary>
/// Interface définissant les propriétés de position d'un
/// salarié dans la hiérarchie de l'entreprise.
/// </summary>

 interface IPositionHierarchie {
  int Indice_Hierarchique {

 }
  double Coeff_Prime {

 }
  DateTime IndiceDepui {

 }
}
```

```
/// <summary>
/// Classe de base abstraite pour le personnel. Cette classe n'est
/// pas instanciable.
/// </summary>

 abstract class Salarie : IPositionHierarchie
 {
  /// attributs identifiant le salarié :
  private int FCodeEmploye ;
  private string FPrenom ;
  private CategoriePerso FCategorie ;
  private string FNom ;
  private string FInsee ;
  protected int FMerite ;
  private int FIndice ;
  private double FCoeffPrime ;
  private double FBasePrime ;
  private DateTime FIndiceDetenu ;

  ///le constructeur de la classe Salarie , payé au mérite :
  public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie,
  string Insee, int Merite, int Indice, double CoeffPrime ) {

  }
  ///le constructeur de la classe Salarie , payé sans mérite :
```

```
 public  Salarie ( ..... )...   {
 }
protected  double  EvaluerPrimeCadreSup ( int  coeffMerite )  {
  return ( 100 + coeffMerite * 8 ) * FCoeffPrime * FBasePrime + FIndice * 7 ;
}
protected  double  EvaluerPrimeCadre ( int  coeffMerite )  {
  return ( 100 + coeffMerite * 6 ) * FCoeffPrime * FBasePrime + FIndice * 5 ;
}
protected  double  EvaluerPrimeMaitrise ( int  coeffMerite )  {
  return ( 100 + coeffMerite * 4 ) * FCoeffPrime * FBasePrime + FIndice * 3 ;
}
protected  double  EvaluerPrimeAgent ( int  coeffMerite )  {
  return ( 100 + coeffMerite * 2 ) * FCoeffPrime * FBasePrime + FIndice * 2 ;
}
 /// propriété abstraite donnant le montant du salaire
/// (virtual automatiquement)
 abstract public  double  MontantPaie   {

 }
 /// propriété identifiant le salarié dans l'entreprise (lecture) :
 public  int  IDentifiant   {

 }
 /// propriété nom du salarié (lecture /ecriture) :
 public  string  Nom   {

 }
 /// propriété prénom du salarié (lecture /ecriture) :
 public  string  Prenom   {

 }
 /// propriété catégorie de personnel du salarié (lecture /ecriture) :
 public  CategoriePerso Categorie   {

 }
 /// propriété n° de sécurité sociale du salarié (lecture /ecriture) :
 public  string  Insee   {

 }
 /// propriété de point de mérite du salarié (lecture /ecriture) ::
 public virtual  int  Merite   {

 }
 /// propriété classement indiciaire dans la hiérarchie (lecture /ecriture) :
 public  int  Indice_Hierarchique   {
  //--lors de l'écriture : Maj de la date de détention du nouvel indice
 }
 /// propriété coefficient de la prime en % (lecture /ecriture) :
 public  double  Coeff_Prime   {

 }
 /// propriété valeur de la prime selon la catégorie (lecture) :
 public  double  Prime   {

 }
 /// date à laquelle l'indice actuel a été obtenu (lecture /ecriture) :
 public  DateTime IndiceDepuis   {

 }
}
```

```
/// <summary>
/// Classe du personnel mensualisé. Implémente la propriété abstraite
/// MontantPaie déclarée dans la classe de base (mère).
/// </summary>

class SalarieMensuel : Salarie
{
/// attributs du salaire annuel :
private double FPrime ;
private double FRemunerationTotal ;

///le constructeur de la classe (salarié au mérite) :
public SalarieMensuel ( int IDentifiant, string Nom, string Prenom,
CategoriePerso Categorie, string Insee, int Merite, int Indice,
double CoeffPrime, double RemunerationTotal )... {
  // la prime est calculée
}
/// implémentation de la propriété donnant le montant du salaire (lecture) :
public .... double MontantPaie {

}
/// propriété de point de mérite du salarié (lecture /ecriture) :
public ... int Merite {

}
}
```

```
/// <summary>
/// Classe du personnel horaire. Implémente la propriété abstraite
/// MontantPaie déclarée dans la classe de base (mère).
/// </summary>

class SalarieHoraire : Salarie
{
/// attributs permettant le calcul du salaire :
private double FPrime ;
private double FTauxHoraire ;
private double FHeuresTravaillees ;

///le constructeur de la classe (salarié non au mérite):
public SalarieHoraire ( int IDentifiant, string Nom, string Prenom,
 CategoriePerso Categorie, string Insee, double TauxHoraire ) ... {

}
/// nombre d'heures effectuées (lecture /ecriture) :
public double HeuresTravaillees {

}
/// implémentation de la propriété donnant le montant du salaire (lecture) :
public override double MontantPaie {

}
}
}
```

Implémenter les classes avec le programme de test suivant :

```
class  ClassUsesSalarie
 {

  static void  InfoSalarie ( SalarieMensuel empl )
  {
   Console .WriteLine ("Employé n°" + empl.IDentifiant + ": " + empl.Nom + " / " + empl.Prenom );
   Console .WriteLine (" n° SS : " + empl.Insee );
   Console .WriteLine (" catégorie : " + empl.Categorie );
   Console .WriteLine (" indice hiérarchique : " + empl.Indice_Hierarchique + " , détenu depuis : " +
empl.IndiceDepuis );
   Console .WriteLine (" coeff mérite : " + empl.Merite );
   Console .WriteLine (" coeff prime : " + empl.Coeff_Prime );
   Console .WriteLine (" montant prime annuelle : " + empl.Prime );
   Console .WriteLine (" montant paie mensuelle: " + empl.MontantPaie );

   double  coefPrimeLoc = empl.Coeff_Prime ;
   int  coefMeriteLoc = empl.Merite ;
   //--impact variation du coef de prime
   for( double  i = 0.5 ;  i < 1 ;  i  += 0.1 )
   {
    empl.Coeff_Prime =  i ;
    Console .WriteLine ("   coeff prime : " + empl.Coeff_Prime );
    Console .WriteLine ("   montant prime annuelle : " + empl.Prime );
    Console .WriteLine ("   montant paie mensuelle: " + empl.MontantPaie );
   }
   Console .WriteLine ("   -----------------------");
   empl.Coeff_Prime = coefPrimeLoc ;
   //--impact variation du coef de mérite
   for( int  i = 0 ;  i < 10 ;  i  ++ )
   {
    empl.Merite =  i ;
    Console .WriteLine ("   coeff mérite : " + empl.Merite );
    Console .WriteLine ("   montant prime annuelle : " + empl.Prime );
    Console .WriteLine ("   montant paie mensuelle: " + empl.MontantPaie );
   }
   empl.Merite = coefMeriteLoc ;
   Console .WriteLine ("=====================================");

  }
  [STAThread]
  static void  Main ( string [] args )
  {
   SalarieMensuel Employe1  =  new  SalarieMensuel ( 123456, "Euton" , "Jeanne" ,
  CategoriePerso.Cadre_Sup, "2780258123456" ,6,700,0.5,50000 );
   SalarieMensuel Employe2  =  new  SalarieMensuel ( 123457, "Yonaize" , "Mah" ,
  CategoriePerso.Cadre, "1821113896452" ,5,520,0.42,30000 );
   SalarieMensuel Employe3  =  new  SalarieMensuel ( 123457, "Ziaire" , "Marie" ,
  CategoriePerso.Maitrise, "2801037853781" ,2,678,0.6,20000 );
   SalarieMensuel Employe4  =  new  SalarieMensuel ( 123457, "Louga" , "Belle" ,
  CategoriePerso.Agent, "2790469483167" ,4,805,0.25,20000 );

   ArrayList ListeSalaries  =  new  ArrayList ();
   ListeSalaries.Add (  Employe1  );
   ListeSalaries.Add (  Employe2  );
   ListeSalaries.Add (  Employe3  );
   ListeSalaries.Add (  Employe4  );
  foreach( SalarieMensuel s  in  ListeSalaries )
```

```
    InfoSalarie ( s );
   Console .WriteLine (">>>  Promotion indice de " + Employe1.Nom + " dans 2 secondes.");
   Thread.Sleep ( 2000 );
   Employe1.Indice_Hierarchique = 710 ;
   InfoSalarie (  Employe1  );
   System .Console.ReadLine ();
  }
 }
```

Résultats obtenus avec le programme de test précédent :

```
Employé n°123456: Euton / Jeanne
 n°SS : 2780258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 700 , détenu depuis : 15/06/2004 19:24:23
 coeff mérite : 6
 coeff prime : 0,5
 montant prime annuelle : 152900
 montant paie mensuelle: 16908,3333333333
   coeff prime : 0,5
   montant prime annuelle : 152900
   montant paie mensuelle: 16908,3333333333
   coeff prime : 0,6
   montant prime annuelle : 182500
   montant paie mensuelle: 19375
   coeff prime : 0,7
   montant prime annuelle : 212100
   montant paie mensuelle: 21841,6666666667
   coeff prime : 0,8
   montant prime annuelle : 241700
   montant paie mensuelle: 24308,3333333333
   coeff prime : 0,9
   montant prime annuelle : 271300
   montant paie mensuelle: 26775
   coeff prime : 1
   montant prime annuelle : 300900
   montant paie mensuelle: 29241,6666666667
   -----------------------
   coeff mérite : 0
   montant prime annuelle : 104900
   montant paie mensuelle: 12908,3333333333
   coeff mérite : 1
   montant prime annuelle : 112900
   montant paie mensuelle: 13575
   coeff mérite : 2
   montant prime annuelle : 120900
   montant paie mensuelle: 14241,6666666667
   coeff mérite : 3
   montant prime annuelle : 128900
   montant paie mensuelle: 14908,3333333333
   coeff mérite : 4
   montant prime annuelle : 136900
   montant paie mensuelle: 15575
   coeff mérite : 5
   montant prime annuelle : 144900
   montant paie mensuelle: 16241,6666666667
   coeff mérite : 6
   montant prime annuelle : 152900
   montant paie mensuelle: 16908,3333333333
   coeff mérite : 7
   montant prime annuelle : 160900
   montant paie mensuelle: 17575
   coeff mérite : 8
   montant prime annuelle : 168900
   montant paie mensuelle: 18241,6666666667
   coeff mérite : 9
```

```
   montant prime annuelle : 176900
   montant paie mensuelle: 18908,3333333333
=====================================
Employé n°123457: Yonaize / Mah
 n°SS : 1821113896452
 catégorie : Cadre
 indice hiérarchique : 520 , détenu depuis : 15/06/2004 19:24:23
 coeff mérite : 5
 coeff prime : 0,42
 montant prime annuelle : 57200
 montant paie mensuelle: 7266,66666666667
```

… tout le tableau :

```
   coeff mérite : …
   montant prime annuelle …
   montant paie mensuelle: …

=====================================
```

…… tous les autres salariés

```
=====================================
>>>  Promotion indice de Euton dans 2 secondes.
Employé n°123456: Euton / Jeanne
 n°SS : 2780258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 710 , détenu depuis : 15/06/2004 19:24:25
 coeff mérite : 6
 coeff prime : 0,5
 montant prime annuelle : 152970
 montant paie mensuelle: 16914,1666666667
   coeff prime : 0,5
   montant prime annuelle : 152970
   montant paie mensuelle: 16914,1666666667
   coeff prime : 0,6
   montant prime annuelle : 182570
   montant paie mensuelle: 19380,8333333333
   coeff prime : 0,7
   montant prime annuelle : 212170
   montant paie mensuelle: 21847,5
   coeff prime : 0,8
   montant prime annuelle : 241770
   montant paie mensuelle: 24314,1666666667
   coeff prime : 0,9
   montant prime annuelle : 271370
   montant paie mensuelle: 26780,8333333333
   coeff prime : 1
   montant prime annuelle : 300970
   montant paie mensuelle: 29247,5
   ------------------------
   coeff mérite : 0
   montant prime annuelle : 104970
   montant paie mensuelle: 12914,1666666667
   coeff mérite : 1
   montant prime annuelle : 112970
   montant paie mensuelle: 13580,8333333333
   coeff mérite : 2
   montant prime annuelle : 120970
   montant paie mensuelle: 14247,5
   coeff mérite : 3
   montant prime annuelle : 128970
   montant paie mensuelle: 14914,1666666667
   coeff mérite : 4
   montant prime annuelle : 136970
   montant paie mensuelle: 15580,8333333333
   coeff mérite : 5
   montant prime annuelle : 144970
   montant paie mensuelle: 1
```

Programme C# solution

Les classes et interface de base :

```
using System ;
using System.Collections ;
using System.Threading ;

namespace cci
{
 enum CategoriePerso { Cadre_Sup,Cadre,Maitrise,Agent,Autre }

 /// <summary>
/// Interface définissant les propriétés de position d'un
/// salarié dans la hiérarchie de l'entreprise.
/// </summary>

 interface IPositionHierarchie
 {
  int Indice_Hierarchique {
   get ;
   set ;
 }
  double Coeff_Prime {
   get ;
   set ;
 }
  DateTime IndiceDepuis {
   get ;
   set ;
 }
}
 /// <summary>
/// Classe de base abstraite pour le personnel. Cette classe n'est
/// pas instanciable.
/// </summary>

 abstract class Salarie : IPositionHierarchie
 {
  /// attributs identifiant le salarié :
  private int FCodeEmploye ;
  private string FPrenom ;
  private CategoriePerso FCategorie ;
  private string FNom ;
  private string FInsee ;
  protected int FMerite ;
  private int FIndice ;
  private double FCoeffPrime ;
  private double FBasePrime ;
  private DateTime FIndiceDetenu ;

  ///le constructeur de la classe employé au mérite :
  public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie,
  string Insee, int Merite, int Indice, double CoeffPrime ) {
   FCodeEmploye = IDentifiant ;
   FNom = Nom ;
   FPrenom = Prenom ;
   FCategorie = Categorie ;
   FInsee = Insee ;
```

```
  FMerite = Merite ;
  FIndice = Indice ;
  FCoeffPrime = CoeffPrime ;
  FIndiceDetenu = DateTime.Now ;
 switch ( FCategorie )
 {
  case CategoriePerso.Cadre_Sup :
   FBasePrime = 2000 ;
  break;
  case CategoriePerso.Cadre :
   FBasePrime = 1000 ;
  break;
  case CategoriePerso.Maitrise :
   FBasePrime = 500 ;
  break;
  case CategoriePerso.Agent :
   FBasePrime = 200 ;
  break;
 }
}
 ///le constructeur de la classe employé sans mérite :
 public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie, string Insee ):
              this( IDentifiant, Nom, Prenom, Categorie, Insee, 0, 0, 0 ) {
}
protected double EvaluerPrimeCadreSup ( int coeffMerite ) {
 return ( 100 + coeffMerite * 8 ) * FCoeffPrime * FBasePrime + FIndice * 7 ;
}
protected double EvaluerPrimeCadre ( int coeffMerite ) {
 return ( 100 + coeffMerite * 6 ) * FCoeffPrime * FBasePrime + FIndice * 5 ;
}
protected double EvaluerPrimeMaitrise ( int coeffMerite ) {
 return ( 100 + coeffMerite * 4 ) * FCoeffPrime * FBasePrime + FIndice * 3 ;
}
protected double EvaluerPrimeAgent ( int coeffMerite ) {
 return ( 100 + coeffMerite * 2 ) * FCoeffPrime * FBasePrime + FIndice * 2 ;
}
 /// propriété abstraite donnant le montant du salaire
/// (virtual automatiquement)
 abstract public double MontantPaie {
  get ;
}
 /// propriété identifiant le salarié dans l'entreprise :
 public int IDentifiant {
  get { return FCodeEmploye ; }
}
 /// propriété nom du salarié :
 public string Nom {
  get { return FNom ; }
  set { FNom = value ; }
}
 /// propriété prénom du salarié :
 public string Prenom {
  get { return FPrenom ; }
  set { FPrenom = value ; }
}
 /// propriété catégorie de personnel du salarié :
 public CategoriePerso Categorie {
  get { return FCategorie ; }
  set { FCategorie = value ; }
}
```

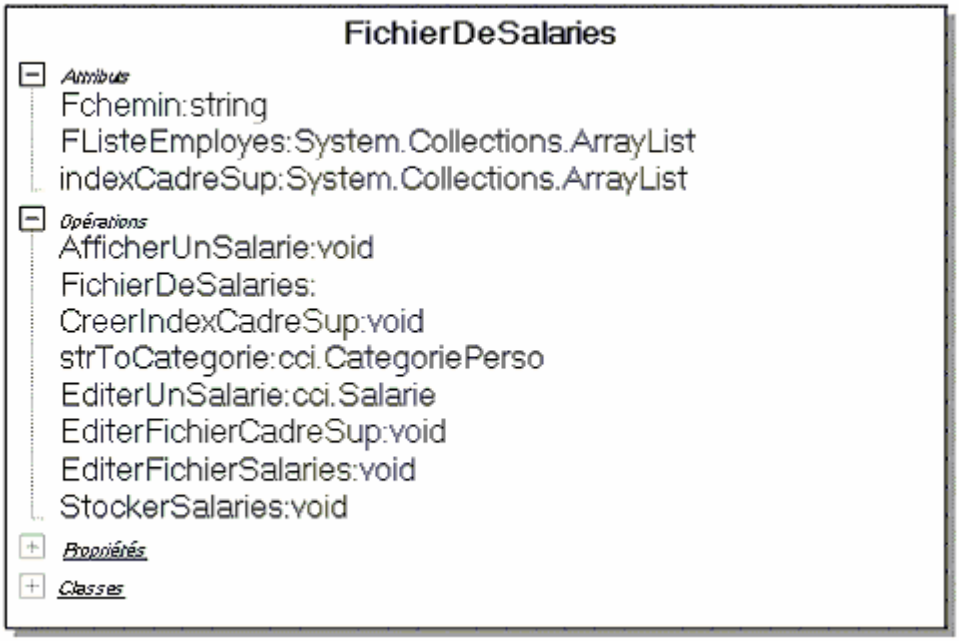
```
 /// propriété n&deg; de sécurité sociale du salarié :
 public string Insee {
  get { return FInsee ; }
  set { FInsee = value ; }
 }
 /// propriété de point de mérite du salarié :
 public virtual int Merite {
  get { return FMerite ; }
  set { FMerite = value ; }
 }
 /// propriété classement indiciaire dans la hiérarchie :
 public int Indice_Hierarchique {
  get { return FIndice ; }
  set {
   FIndice = value ;
   //--Maj de la date de détention du nouvel indice :
   IndiceDepuis = DateTime.Now ;
  }
 }
 /// propriété coefficient de la prime en %:
 public double Coeff_Prime {
  get { return FCoeffPrime ; }
  set { FCoeffPrime = value ; }
 }
 /// propriété valeur de la prime :
 public double Prime {
  get {
   switch ( FCategorie )
   {
    case CategoriePerso.Cadre_Sup :
      return EvaluerPrimeCadreSup ( FMerite );
    case CategoriePerso.Cadre :
      return EvaluerPrimeCadre ( FMerite );
    case CategoriePerso.Maitrise :
      return EvaluerPrimeMaitrise ( FMerite );
    case CategoriePerso.Agent :
      return EvaluerPrimeAgent ( FMerite );
    default :
      return EvaluerPrimeAgent ( 0 );
   }
  }
 }
 /// date è laquelle l'indice actuel a été obtenu :
 public DateTime IndiceDepuis {
  get { return FIndiceDetenu ; }
  set { FIndiceDetenu = value ; }
 }
}

 /// <summary>
/// Classe du personnel mensualisé. Implémente la propriété abstraite
/// MontantPaie déclarée dans la classe de base (mère).
/// </summary>

 class SalarieMensuel : Salarie
 {
  /// attributs du salaire annuel :
  private double FPrime ;
  private double FRemunerationTotal ;
```

```
  ///le constructeur de la classe (salarié au mérite) :
  public SalarieMensuel ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie,
        string Insee, int Merite, int Indice, double CoeffPrime, double RemunerationTotal ):
              base ( IDentifiant, Nom, Prenom, Categorie, Insee, Merite, Indice, CoeffPrime ) {
   FPrime = this .Prime ;
   FRemunerationTotal = RemunerationTotal ;
  }
  /// implémentation de la propriété donnant le montant du salaire :
  public override double MontantPaie {
   get { return ( FRemunerationTotal + this .Prime ) / 12 ; }
  }
  /// propriété de point de mérite du salarié :
  public override int Merite {
   get { return FMerite ; }
   set { FMerite = value ;  FPrime = this .Prime ; }
  }
 }

 /// <summary>
/// Classe du personnel horaire. Implemente la propriété abstraite
/// MontantPaie déclarée dans la classe de base (mère).
/// </summary>

 class SalarieHoraire : Salarie
 {
  /// attributs permettant le calcul du salaire :
  private double FPrime ;
  private double FTauxHoraire ;
  private double FHeuresTravaillees ;

  ///le constructeur de la classe (salarié non au mérite):
  public SalarieHoraire ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie,
        string Insee, double TauxHoraire ) :
              base ( IDentifiant, Nom,Prenom, Categorie, Insee ) {
   FTauxHoraire = TauxHoraire ;
   FHeuresTravaillees = 0 ;
   FPrime = 0 ;
  }
  /// nombre d'heures effectuées :
  public double HeuresTravaillees {
   get { return FHeuresTravaillees ; }
   set { FHeuresTravaillees = value ; }
  }
  /// implémentation de la propriété donnant le montant du salaire :
  public override double MontantPaie {
   get { return FHeuresTravaillees * FTauxHoraire + FPrime ; }
  }
 }
}
```

Classes, objet et IHM

Classe de salariés dans un fichier de l'entreprise

Nous reprenons les trois classes de l'exercice précédent définissant un salarié horaire et mensualisé dans une entreprise. Nous créons un fichier des salariés de l'entreprise, pour cela nous définissons une classe **FichierDeSalaries** permettant de gérer le fichier des salariés de l'entreprise :



Soit le squelette de la classe **FichierDeSalaries** :

```
using System ;
using System.Collections ;
using System.Threading ;
using System.IO ;

namespace cci
{
 class FichierDeSalaries
 {
  private string Fchemin ;
  private ArrayList FListeEmployes ; // liste des nouveaux employés à entrer dans le fichier
  private ArrayList indexCadreSup ; // Table d'index des cadres supérieurs du fichier

  // méthode static affichant un objet Salarie à la console :
  public static void AfficherUnSalarie ( Salarie Employe ) {
   // pour l'instant un salarié mensualisé seulement
 }
  // constructeur de la classeFichierDeSalaries
  public FichierDeSalaries ( string chemin, ArrayList Liste ) {

 }
  // méthode de création de la table d'index des cadre_sup :
  public void CreerIndexCadreSup ( ) {
```

```
}
 // méthode convertissant le champ string catégorie en la constante enum associée
 private  CategoriePerso strToCategorie ( string  s )  {

 }
 // méthode renvoyant un objet SalarieMensuel de rang fixé dans le fichier
 private  Salarie EditerUnSalarie ( int  rang )  {
  SalarieMensuel  perso ;
   ...........
  perso  =  new  SalarieMensuel ( IDentifiant, Nom, Prenom, Categorie, Insee,
            Merite, Indice, CoeffPrime, RemunerationTotal );
   ...........
  return  perso ;
}
 // méthode affichant sur la console à partir de la table d'index :
 public void  EditerFichierCadreSup ( )
{
  ...........
 foreach( int  ind  in  indexCadreSup )
 {
   AfficherUnSalarie (  EditerUnSalarie ( ind ) );
 }
  ...........
}
 // méthode affichant sur la console le fichier de tous les salariés :
 public void  EditerFichierSalaries ( )  {

 }
 // méthode créant et stockant des salariés dans le fichier :
 public void  StockerSalaries ( ArrayList ListeEmploy )
{
  ...........
  // si le fichier n'existe pas => création du fichier sur disque :
   fichierSortie  =  File.CreateText ( Fchemin );
   fichierSortie.WriteLine ("Fichier des personnels");
   fichierSortie.Close ( );
 ...........
  // ajout dans le fichier de toute  la liste  :
  ...........
  foreach( Salarie s  in  ListeEmploy )
  {

  }
 ...........
 }
}
```

Implémenter la classe FichierDeSalaries avec le programme de test suivant :

```
class  ClassUsesSalarie
{
 /// <summary>
/// Le point d'entrée principal de l'application.
/// </summary>
 static void  InfoSalarie ( SalarieMensuel empl )
{
  FichierDeSalaries.AfficherUnSalarie ( empl );
  double  coefPrimeLoc = empl.Coeff_Prime ;
```

```
 int  coefMeriteLoc = empl.Merite ;
 //--impact variation du coef de prime
 for( double  i = 0.5 ;  i < 1 ;  i  += 0.1 )
 {
  empl.Coeff_Prime =  i ;
  Console .WriteLine ("   coeff prime : " + empl.Coeff_Prime );
  Console .WriteLine ("   montant prime annuelle : " + empl.Prime );
  Console .WriteLine ("   montant paie mensuelle: " + empl.MontantPaie );
 }
 Console .WriteLine ("   -----------------------");
 empl.Coeff_Prime = coefPrimeLoc ;
 //--impact variation du coef de mérite
 for( int  i = 0 ;  i < 10 ;  i  ++ )
 {
  empl.Merite =  i ;
  Console .WriteLine ("   coeff mérite : " + empl.Merite );
  Console .WriteLine ("   montant prime annuelle : " + empl.Prime );
  Console .WriteLine ("   montant paie mensuelle: " + empl.MontantPaie );
 }
 empl.Merite = coefMeriteLoc ;
 Console .WriteLine ("=====================================");

}
 [STAThread]
 static void  Main ( string [ ] args )
{
 SalarieMensuel Employe1  =  new  SalarieMensuel ( 123456, "Euton" , "Jeanne" ,
     CategoriePerso.Cadre_Sup, "2780258123456" ,6,700,0.5,50000 );
 SalarieMensuel Employe2  =  new  SalarieMensuel ( 123457, "Yonaize" , "Mah" ,
     CategoriePerso.Cadre, "1821113896452" ,5,520,0.42,30000 );
 SalarieMensuel Employe3  =  new  SalarieMensuel ( 123458, "Ziaire" , "Marie" ,
     CategoriePerso.Maitrise, "2801037853781" ,2,678,0.6,20000 );
 SalarieMensuel Employe4  =  new  SalarieMensuel ( 123459, "Louga" , "Belle" ,
     CategoriePerso.Agent, "2790469483167" ,4,805,0.25,20000 );

 ArrayList ListeSalaries  =  new  ArrayList ();
 ListeSalaries.Add (  Employe1  );
 ListeSalaries.Add (  Employe2  );
 ListeSalaries.Add (  Employe3  );
 ListeSalaries.Add (  Employe4  );
 foreach( SalarieMensuel s  in  ListeSalaries )
  InfoSalarie ( s );
 Console .WriteLine (">>>  Promotion indice de " + Employe1.Nom + " dans 2 secondes.");
 Thread.Sleep ( 2000 );
 Employe1.Indice_Hierarchique = 710 ;
 InfoSalarie (  Employe1  );
 //-----------------------------------------//
 FichierDeSalaries Fiches  =  new  FichierDeSalaries ("fichierSalaries.txt" ,ListeSalaries );
 Console .WriteLine (">>>  Attente 3 s pour création de nouveaux salariés");
 Thread.Sleep ( 3000 );
 Employe1  =  new  SalarieMensuel ( 123460, "Miett" , "Hamas" ,
     CategoriePerso.Cadre_Sup, "1750258123456" ,4,500,0.7,42000 );
 Employe2  =  new  SalarieMensuel ( 123461, "Kong" , "King" ,
     CategoriePerso.Cadre, "1640517896452" ,4,305,0.62,28000 );
 Employe3  =  new  SalarieMensuel ( 123462, "Zaume" , "Philippo" ,
     CategoriePerso.Maitrise, "1580237853781" ,2,245,0.8,15000 );
 Employe4  =  new  SalarieMensuel ( 123463, "Micoton" , "Mylène" ,
     CategoriePerso.Agent, "2850263483167" ,4,105,0.14,12000 );
 ListeSalaries  =  new  ArrayList ();
 ListeSalaries.Add (  Employe1  );
```

```
    ListeSalaries.Add (  Employe2  );
    ListeSalaries.Add (  Employe3  );
    ListeSalaries.Add (  Employe4  );
    Fiches.StockerSalaries ( ListeSalaries );
    Fiches.EditerFichierSalaries ( );
    Fiches.CreerIndexCadreSup ( );
    Fiches.EditerFichierCadreSup ( );
    System .Console.ReadLine ( );
  }
}
```

Pour tester le programme précédent, on donne le fichier des salariés **fichierSalaries.txt** suivant :

```
Fichier des personnels
123456
Euton
Jeanne
*Cadre_Sup
2780258123456
6
710
15/02/2004 19:52:38
0,5
152970
16914,1666666667
123457
Yonaize
Mah
*Cadre
1821113896452
5
520
15/02/2004 19:52:36
0,42
57200
7266,66666666667
123458
Ziaire
Marie
*Maitrise
2801037853781
2
678
15/02/2004 19:52:36
0,6
34434
4536,16666666667
123459
Louga
Belle
*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,83333333333
123460
Miett
```

```
Hamas
*Cadre_Sup
1750258123456
4
500
15/02/2004 19:52:41
0,7
188300
19191,6666666667
123461
Kong
King
*Cadre
1640517896452
4
305
15/02/2004 19:52:41
0,62
78405
8867,08333333333
123462
Zaume
Philippo
*Maitrise
1580237853781
2
245
15/02/2004 19:52:41
0,8
43935
4911,25
123463
Micoton
Mylène
*Agent
2850263483167
4
105
15/02/2004 19:52:41
0,14
3234
1269,5
==============================
```

La classe C# *FichierDeSalaries* solution

```
using System ;
using System.Collections ;
using System.Threading ;
using System.IO ;

namespace cci
{
 class FichierDeSalaries
  {
  private string Fchemin ;
  private ArrayList FListeEmployes ;
  private ArrayList indexCadreSup ;
```

```
// méthode static affichant un objet Salarie è la console :
public static void  AfficherUnSalarie ( Salarie Employe )  {
if( Employe  is  SalarieMensuel )
{
  SalarieMensuel empl  =  ( Employe  as  SalarieMensuel );
  Console .WriteLine ("Employé n°" + empl.IDentifiant + ": " + empl.Nom + " / " + empl.Prenom );
  Console .WriteLine (" n° SS : " + empl.Insee );
  Console .WriteLine (" catégorie : " + empl.Categorie );
  Console .WriteLine (" indice hiérarchique : " + empl.Indice_Hierarchique + " , détenu depuis : "
                                + empl.IndiceDepuis );
  Console .WriteLine (" coeff mérite : " + empl.Merite );
  Console .WriteLine (" coeff prime : " + empl.Coeff_Prime );
  Console .WriteLine (" montant prime annuelle : " + empl.Prime );
  Console .WriteLine (" montant paie mensuelle: " + empl.MontantPaie );
}
}
// constructeur de la classeFichierDeSalaries
public  FichierDeSalaries (  string  chemin, ArrayList Liste  )  {
 Fchemin = chemin ;
 FListeEmployes  =  Liste ;
 StockerSalaries ( FListeEmployes );
}
// méthode de création de la table d'index des cadre_sup :
public void  CreerIndexCadreSup ( )  {
 // Ouvre le fichier pour le lire
 StreamReader fichierEntree  =  File.OpenText ( Fchemin );
 string  Ligne ;
 int  indexLigne = 0 ;
 indexCadreSup  =  new  ArrayList ( );
while (( Ligne  =  fichierEntree.ReadLine ())  !=  null)
{
  indexLigne ++ ;
 if("*" + CategoriePerso.Cadre_Sup.ToString () == Ligne )
 {
   Console .WriteLine ("++> " + Ligne + " : " + indexLigne );
   indexCadreSup.Add ( indexLigne - 3 );
 }
}
 fichierEntree.Close ();
}
// méthode convertissant le champ string catégorie en la constante enum associée
private  CategoriePerso strToCategorie ( string  s )  {
switch( s )
{
 case "*Cadre_Sup":return  CategoriePerso.Cadre_Sup ;
 case "*Cadre":return  CategoriePerso.Cadre ;
 case "*Maitrise":return  CategoriePerso.Maitrise ;
 case "*Agent":return  CategoriePerso.Agent ;
 case "*Autre":return  CategoriePerso.Autre ;
 default : return  CategoriePerso.Autre ;
}
}
// méthode renvoyant un objet SalarieMensuel de rang fixé dans le fichier
private  Salarie EditerUnSalarie ( int  rang )  {
 int  compt = 0 ;
 string  Ligne ;

 int  IDentifiant = 0 ;
 string  Nom = "" , Prenom = "";
```

```
 CategoriePerso Categorie = CategoriePerso.Autre ;
 string  Insee = "";
 int  Merite = 0,  Indice = 0 ;
 DateTime delai =  DateTime.Now ;
 double  CoeffPrime = 0, RemunerationTotal = 0, MontantPaie = 0 ;
 SalarieMensuel perso ;
 StreamReader f  =  File.OpenText ( Fchemin );
 //System .IFormatProvider format  =  new  System .Globalization.CultureInfo ("fr-FR" , true );

 while (( Ligne  =  f.ReadLine ( ))  !=  null)
 {
  compt ++ ;
  if ( compt == rang )
  {
   IDentifiant  =  Convert.ToInt32 ( Ligne );
   Nom  =  f.ReadLine ( );
   Prenom  =  f.ReadLine ( );
   Categorie  =   strToCategorie ( f.ReadLine ( ));
   Insee  =  f.ReadLine ( );
   Merite  =  Convert.ToInt32 ( f.ReadLine ( ));
   Indice  =  Convert.ToInt32 ( f.ReadLine ( ));
   delai  =  DateTime.Parse ( f.ReadLine ( )  );
   CoeffPrime  =  Convert.ToDouble ( f.ReadLine ( ));
   RemunerationTotal  =  Convert.ToDouble ( f.ReadLine ( ));
   MontantPaie  =  Convert.ToDouble ( f.ReadLine ( ));
  break;
  }
 }
 f.Close ( );
 perso  =  new  SalarieMensuel ( IDentifiant, Nom, Prenom, Categorie, Insee, Merite,
              Indice, CoeffPrime, RemunerationTotal );
 perso.IndiceDepuis  =  delai ;
 return  perso ;
}
 // méthode affichant sur la console è partir de la table d'index :
 public void  EditerFichierCadreSup ( )  {
  StreamReader fichierEntree  =  File.OpenText ( Fchemin );
  if( indexCadreSup == null)
   CreerIndexCadreSup ();

 foreach( int  ind  in  indexCadreSup )
 {
  AfficherUnSalarie (  EditerUnSalarie ( ind ) );
 }
  fichierEntree.Close ( );
}
 // méthode affichant sur la console le fichier de tous les salariés :
 public void  EditerFichierSalaries ( )  {
  // Ouvre le fichier pour le lire
  StreamReader fichierEntree  =  File.OpenText ( Fchemin );
  string  Ligne ;
 while (( Ligne  =  fichierEntree.ReadLine ( ))  !=  null)
 {
   Console .WriteLine ( Ligne );
 }
  fichierEntree.Close ( );
}
 // méthode créant et stockant des salariés dans le fichier :
 public void  StockerSalaries ( ArrayList ListeEmploy )  {
  StreamWriter fichierSortie ;
```

```
  if ( ! File.Exists ( Fchemin ))
  {
   // création du fichier sur disque :
   fichierSortie  =  File.CreateText ( Fchemin );
   fichierSortie.WriteLine ("Fichier des personnels");
   fichierSortie.Close ( );
  }

  // ajout dans le fichier de tout le  :
  fichierSortie  =  File.AppendText ( Fchemin );
 if ( FListeEmployes.Count  != 0 )
  foreach( Salarie s  in  ListeEmploy )
  {
    fichierSortie.WriteLine ( s.IDentifiant );
    fichierSortie.WriteLine ( s.Nom );
    fichierSortie.WriteLine ( s.Prenom );
    fichierSortie.WriteLine ( '*'+ s.Categorie.ToString ( ));
    fichierSortie.WriteLine ( s.Insee );
   if( s  is  SalarieMensuel )
   {
     SalarieMensuel sLoc  =  ( s  as  SalarieMensuel );
     fichierSortie.WriteLine ( sLoc.Merite );
     fichierSortie.WriteLine ( sLoc.Indice_Hierarchique );
     fichierSortie.WriteLine ( sLoc.IndiceDepuis );
     fichierSortie.WriteLine ( sLoc.Coeff_Prime );
     fichierSortie.WriteLine ( sLoc.Prime );
   }
   else
     fichierSortie.WriteLine (( s  as  SalarieHoraire ) .HeuresTravaillees );
    fichierSortie.WriteLine ( s.MontantPaie );
  }
  fichierSortie.Close ( );
 }
}
```

Classes, objet et IHM

Construction d'un ensemble de caractères

Soit à construire une classe **setOfChar** ensemble de caractères possédant certaines caractéristiques de base d'un ensemble, il est demandé que les opérations suivantes soient présentes : ajouter, enlever un élément de l'ensemble, test d'appartenance d'un élément à l'ensemble, cardinal de l'ensemble (la redondance est acceptée)

Il est aussi demandé à ce que l'ensemble propose deux événements OnInserer qui se produit lorsque l'on ajoute un nouvel élément à l'ensemble et OnEnlever qui a lieu lorsque l'on supprime un élément de l'ensemble.

La classe **setOfChar** héritera de la classe **CollectionBase** et implémentera une interface événementielle **IEventEnsemble** : **class** setOfChar **:** CollectionBase, IeventEnsemble.

CollectionBase est une classe de .Net Framework et fournit la classe de base abstract pour une collection fortement typée :

> **System.Object**
> **|__System.Collections.CollectionBase**
>
> **public abstract class CollectionBase : IList, ICollection, IEnumerable**

L'interface **IList** représente une liste d'objets accessibles séparément par indexeur et des méthodes classiques de gestion de liste dont nous extrayons ci-dessous les principales utiles, à l'exercice à traiter:

Méthodes publiques

int Add(object ***valeur*** **);**	Ajoute l'élément ***valeur*** dans la liste
void Remove(object ***valeur*** **);**	Enlève l'élément ***valeur*** dans la liste
bool Contains(object ***valeur*** **);**	La liste contient l'élément ***valeur***.

IeventEnsemble est une interface qui est donnée pour décrire les deux événements auxquels un ensemble doit être sensible :

```
interface IEventEnsemble {
  event EventHandler OnInserer;
  event EventHandler OnEnlever;
 }
```

Question :

compléter dans le squelette de programme ci-après, la classe ensemble de caractère **setOfChar.**

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

```
namespace cci
{

interface IEventEnsemble {
  event EventHandler OnInserer;
  event EventHandler OnEnlever;
 }

public class setOfChar : CollectionBase, IEventEnsemble   {

 public event EventHandler OnInserer;
 public event EventHandler OnEnlever;

 public static setOfChar operator + ( setOfChar e1, setOfChar e2)  {
   // surcharge de l'opérateur d'addition étendu aux ensembles de char
  ....
 }

 //-- les constructeurs de la classe servent à initialiser l'ensemble :
 public setOfChar( )  {
   ....
 }

 public setOfChar(string s) : this(s.ToCharArray( ))  {
   ....
 }

 public setOfChar(char[ ]  t)  {
   ....
 }

 public char[ ] ToArray( )  {
  // renvoie les éléments de l'ensemble sous forme d'un tableau de char
    ....
 }

 public override string ToString ( )  {
 // renvoie les éléments de l'ensemble sous forme d'une chaîne
     ....
 }

 public int Card  {
 // cardinal de l'ensemble
     ....
 }

 protected virtual void Inserer(  object sender,  EventArgs e )   {
  // lance l'événement OnInserer
     ....
 }

 protected virtual void Enlever(  object sender,  EventArgs e )   {
  // lance l'événement OnEnlever
   ....
 }

 public char this[ int index ]    {
  // indexeur de l'ensemble
    ....
```

```
}

public int Add( char value )   {
 // ajoute un élément à l'ensemble
    ....
}

public void Remove( char value )   {
 // enlève un élément à l'ensemble s'il est présent
    ....
}

public bool Contains( char value )   {
 // true si value est dans type setOfChar, false sinon.
    ....
}

protected override void OnInsert( int index, Object value )   {
    ....
}

protected override void OnRemove( int index, Object value )   {
    ....
}
}
```

La méthode **protected virtual** void **OnInsert** est présente dans la classe **CollectionBase**, elle sert à exécuter des actions avant l'insertion d'un nouvel élément dans la collection. Elle va servir de base au lancement de l'événement **OnInserer**.

La méthode **protected virtual** void **OnRemove** est présente dans la classe **CollectionBase**, elle sert à exécuter des actions lors de la suppression d'un élément dans la collection. Elle va servir de base au lancement de l'événement **OnEnlever**.

Classe permettant de tester le fonctionnement de setOfChar

```
public class UtiliseSetOfChar
{

 private static void EnsCarOnEnlever( object sender,  EventArgs e )    {
  Console.WriteLine();
  Console.WriteLine("Evénément - on va enlever un élément de : "+sender.ToString());
  Console.WriteLine();
 }

 public static void Main( )   {

  // initialisation
  setOfChar EnsCar1 = new setOfChar();
  setOfChar EnsCar,EnsCar2;

  // ajout d'éléments
  EnsCar1.Add('a');
  EnsCar1.Add('b');
  EnsCar1.Add('c');
  EnsCar1.Add('d');
  EnsCar1.Add('e');
  Console.WriteLine("card="+EnsCar1.Card+" ; "+EnsCar1.ToString());
  EnsCar2=new setOfChar("xyztu#");
  Console.WriteLine("card="+EnsCar2.Card+" ; "+EnsCar2.ToString());
  EnsCar=EnsCar1+EnsCar2;
  EnsCar.OnEnlever += new EventHandler(EnsCarOnEnlever);
  Console.WriteLine("card="+EnsCar.Card+" ; "+EnsCar.ToString());
  Console.WriteLine();

  // Contenu de l'ensemble avec for
  Console.WriteLine( "Contenu de l'ensemble:" );
```

```
  for(int i=0; i<EnsCar.Card; i++)
   Console.Write( EnsCar[i]+"," );
  Console.WriteLine();
  // on enlève un élément dans l'ensemble
  EnsCar.Remove( 'd' );

  // Contenu de l'ensemble avec foreach
  Console.WriteLine( "Contenu de l'ensemble après enlèvement de l'élément d :" );
  foreach(char elt in EnsCar)
   Console.Write(elt+"," );
  Console.ReadLine();
 }
}
}
```

Résultats de l'exécution :

```
card=5 ; abcde
card=6 ; xyztu#
card=11 ; abcdexyztu#

Contenu de l'ensemble:
a,b,c,d,e,x,y,z,t,u,#,

Evénément - on va enlever un élément de : abcdexyztu#

Contenu de l'ensemble après enlèvement de l'élément d :
a,b,c,e,x,y,z,t,u,#,
```

La classe C# *setOfChar* solution

```
public class setOfChar : CollectionBase, IEventEnsemble
{

 public event EventHandler OnInserer;
 public event EventHandler OnEnlever;

 public static setOfChar operator + ( setOfChar e1, setOfChar e2) {
   return new setOfChar(e1.ToString( )+e2.ToString( ));
 }

 public setOfChar( )
 {  }

 public setOfChar(string s) : this(s.ToCharArray( ))
 {   }

 public setOfChar(char[ ] t)   {
   foreach(char car in t)
    this.Add(car);
  }

  public char[ ] ToArray( )
  {
   char[ ] t = new char[this.Count];
   for(int i=0; i<this.Count; i++)
    t[i]=this[i];
   return t;
  }
```

On convertit les deux ensembles sous forme de chaînes, on les concatène, puis on appelle le deuxième constructeur. On peut maintenant écrire :
E = F+G où E, F, G sont des **setOfChar**.

Ce constructeur permet de charger l'ensemble par une chaîne **s** en appelant le constructeur du dessous par conversion de la chaîne **s** en tableau de char (méthode ToCharArray).

Le seul constructeur qui a un corps et permettant de charger l'ensemble avec un tableau de caractères.

```
    public override string ToString( )   {
     return new string(this.ToArray( ));
    }

    public int Card   {
     get    {
      return this.Count;
     }
    }

    protected virtual void Inserer(  object sender,  EventArgs e )    {
      if ( OnInserer != null )
        OnInserer( sender , e );
    }

    protected virtual void Enlever(  object sender,  EventArgs e )    {
      if ( OnEnlever != null )
        OnEnlever( sender , e );
    }

    public char this[ int index ]     {
     get     {
      return( (char) List[index] );
     }
     set     {
      List[index] = value;
     }
    }

    public int Add( char value )     {
      return ( List.Add( value ) );
    }

    public void Remove( char value )     {
      if (Contains(value))
        List.Remove( value );
    }

    public bool Contains( char value )     {
      return ( List.Contains( value ) );
    }

    protected override void OnInsert ( int index, Object value )     {
       Inserer( this , EventArgs.Empty );
    }

    protected override void OnRemove( int index, Object value )     {
       Enlever ( this , EventArgs.Empty );
    }
  }
```

OnInsert appelle la méthode Inserer qui lance l'éventuel gestionnaire d'événement **OnInserer**.

OnRemove appelle la méthode Enlever qui lance l'éventuel gestionnaire d'événement **OnEnlever**.

Si l'on veut éliminer la redondance d'élément (un élément n'est présent qu'une seule fois dans un ensemble) il faut agir sur la méthode d'ajout (add) et vérifier que l'ajout est possible.

L'ajout est possible si l'élément à ajouter n'est pas déjà contenu dans la liste :

Comme la méthode add renvoie le rang d'insertion de l'élément, nous lui faisons renvoyer la valeur -1 lorsque l'élément n'est pas ajouté parce qu'il est déjà présent dans la liste :

```
public int Add( char value )    {
  if (!Contains(value) )
     return ( List.Add ( value ) );
  else
     return -1;
  }
```

Nous pouvons aussi prévoir d'envoyer un message à l'utilisateur de la classe sous forme d'une fenêtre de dialogue l'avertissant du fait qu'un élément était déjà présent et qu'il n'a pas été rajouté. Nous utilisons la classe **MessageBox** de C# qui sert à afficher un message pouvant contenir du texte, des boutons et des symboles :

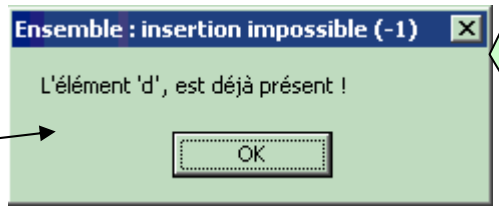
> **System.Object**
> **|__ System.Windows.Forms.MessageBox**
>
> Plus particulièrement, nous utilisons une des surcharges de la méthode **Show** :
>
> **public static** DialogResult **Show(** string ***TexteMess*** , string ***caption*** **);**

Code de la méthode Add (avec message)

```
public int Add( char value )    {
  if (!Contains(value) )
     return ( List.Add ( value ) );
  else {
          MessageBox.Show ("L'élément  ' "+value.ToString( ) +" ' , est déjà présent !" ,
                  "Ensemble : insertion impossible (-1)") ;
          return -1;
       }
}
```

Voici dans cette éventualité, ce que provoque l'exécution de la sixième ligne du code ci-dessous sur la deuxième demande d'insertion de l'élément 'd' dans l'ensemble EnsCar1 :

```
EnsCar1.Add('a');
EnsCar1.Add('b');
EnsCar1.Add('c');
EnsCar1.Add('d');
EnsCar1.Add('e');
EnsCar1.Add('d');
```



Classes, objet et IHM

Construction d'une classe d'ensemble générique

Soit à construire une classe **setOfObject** d'ensemble plus générale que celle de l'exercice précédent. Nous souhaitons en, effet disposer d'une classe de type ensemble possédant les fonctionnalités de la classe **setOfChar** (ajouter, enlever un élément de l'ensemble, test d'appartenance d'un élément à l'ensemble, cardinal de l'ensemble, non redondance d'un élément), qui puisse accueillir des éléments de même type mais que ce type puisse être n'importe quel type héritant de la classe object.

Cette classe d'ensemble, proposera deux événements OnInserer qui se produit lorsque l'on ajoute un nouvel élément à l'ensemble et OnEnlever qui a lieu lorsque l'on supprime un élément de l'ensemble.

La classe **setOfObject** héritera de la classe **CollectionBase** et implémentera une interface événementielle **IEventEnsemble** : **class** setOfObject **:** CollectionBase, IEventEnsemble

Conseils :

> Par rapport à l'exercice précédent, il faut faire attention à l'ajout d'un élément du même type que tous ceux qui sont déjà présents dans l'ensemble et refuser un nouvel élément qui n'est pas strictement du même type. Il faut donc utiliser le mécanisme de reflexion de C# (connaître le type dynamique d'un objet lors de l'exécution) contenu dans la classe Type.
>
> Nous proposons un squelette détaillé de la classe **setOfObject** à compléter, les méthodes qui sont identiques à celle de l'exercice précédent ont été mise avec leur code, les autres sont à définir par le lecteur.

Nous souhaitons disposer de plusieurs surcharges du constructeur d'ensemble générique :

constructeur	fonctionnalité
public setOfObject**( )**	Construit un ensemble vide *(de n'importe quel type)*
public setOfObject **(**object[ ] t**)**	Rempli un ensemble à partir d'un tableau d'object. *(le type des éléments de l'ensemble construit est automatiquement celui du premier élément du* object[ ]*)*
public setOfObject **(**Array t**)**	Rempli un ensemble à partir d'un tableau de type Array. *(le type des éléments de l'ensemble construit est automatiquement celui du premier élément du* Array *)*
public setOfObject **(**ArrayList t**)**	Rempli un ensemble à partir d'un tableau de type ArrayList. *(le type des éléments de l'ensemble construit est automatiquement celui du premier élément du* ArrayList *)*
public setOfObject **(**string s**)**	Rempli un ensemble à partir chaîne. *(le type des éléments de l'ensemble construit est automatiquement celui du premier élément, ici* **char** *)*

Nous proposons enfin, de prévoir un champ protégé nommé FtypeElement qui contient le type de l'élément de l'ensemble qui peut varier au cours du temps Car lorsque l'ensemble est vide il n'a pas de type d'élément c'est l'ajout du premier élément qui détermine le type de l'ensemble et donc des futurs autres éléments à introduire. Ce champ protégé devra être accessible en lecture seulement par tout utilisateur de la classe.

Code à compléter :

```
public class setOfObject : CollectionBase, IEventEnsemble  {

 protected Type FTypeElement;

 public Type TypeElement {
  get  {
   return FTypeElement;
  }
 }

 public event EventHandler OnInserer;
 public event EventHandler OnEnlever;

 public static setOfObject operator + ( setOfObject e1, setOfObject e2) {
  …..
 }

 public setOfObject() {
 }

 public setOfObject(object[] t) {
  …..
 }

 public setOfObject(Array t) {
  …..
 }

 public setOfObject (ArrayList t) ….. {
 }

 public setOfObject(string s) {
  …..
 }

 public virtual object[] ToArray() {
  …..
 }

 public override string ToString() {
  …..
 }

 public int Card {
  get   {
   return this.Count;
  }
 }

 protected virtual void Inserer(  object sender,  EventArgs e )  {
  if ( OnInserer != null )
   OnInserer( sender , e );
 }

 protected virtual void Enlever(  object sender,  EventArgs e )  {
  if ( OnEnlever != null )
   OnEnlever( sender , e );
 }
```

```
public object this[ int index ]  {
 get   {
  return( List[index] );
 }
 set   {
  List[index] = value;
 }
}

public int Add( object value )  {
 /* ajoute un élément à l'ensemble sinon renvoie-1 si déjà présent
ou bien renvoie -2 si l'élément n'est pas du même type que les autres */
    …..
}

public void Remove( object value )  {
 // enlève un élément à l'ensemble s'il est présent et de même type
   …..
}

public bool Contains( object value )  {
 // true si value est dans type setOfChar, false sinon.
 return( List.Contains( value ) );
}

protected override void OnInsert( int index, Object value )  {
    Inserer( this , EventArgs.Empty );
}

protected override void OnRemove( int index, Object value )  {
   Enlever( this , EventArgs.Empty );
 }
}
```

La classe C# *setOfObject* solution

```
public class setOfObject : CollectionBase, IEventEnsemble  {

 protected Type FTypeElement;

 public Type TypeElement {
  get  {
   return FTypeElement;
  }
 }

 public event EventHandler OnInserer;
 public event EventHandler OnEnlever;

 public static setOfObject operator + ( setOfObject e1, setOfObject e2) {
  ArrayList t = new ArrayList(e1);
  foreach(object elt in e2)
     t.Add(elt);
  return new setOfObject(t);
 }
```

Champ FtypeElement contenant le type de l'ensemble.

Propriété permettant de lire le champ FtypeElement contenant le type de l'ensemble.

On se sert du constructeur fondé sur le remplissage à partir d'un ArrayList.

```
public setOfObject ( ) {
 FTypeElement = null;
}

public setOfObject(object[ ] t) {
 foreach(object elt in t)
  this.Add(elt);
}

public setOfObject(Array t) {
 foreach(object elt in t)
  this.Add(elt);
}

public setOfObject(ArrayList t) : this(t.ToArray( ) ) {
}

public setOfObject(string s) {
 char[ ] t = s.ToCharArray( );
 foreach(char elt in t)
  this.Add(elt);
}

public virtual object[ ] ToArray( ) {
 object[ ] t = new object[this.Count];
 for(int i=0; i<this.Count; i++)
  t[i]=this[i];
 return t;
}

public override string ToString( ) {
   return Convert.ToString(this.ToArray());
}

public int Card {
 get   {
          return this.Count;
        }
}

protected virtual void Inserer(  object sender,  EventArgs e )  {
 if ( OnInserer != null )
  OnInserer( sender , e );
}

protected virtual void Enlever(  object sender,  EventArgs e )  {
 if ( OnEnlever != null )
  OnEnlever( sender , e );
}

public object this[ int index ]   {
 get   {
          return( List[index] );
        }
 set   {
          List[index] = value;
        }
}
```

Le constructeur vide initialise le type de l'ensemble.

Le constructeur ajoute tous les objets du tableau t dans l'ensemble.

Le constructeur ajoute tous les objets du tableau t dans l'ensemble.

Le constructeur convertit ArrayList t en object[ ] t et appelle le constructeur setOfObject(object[ ] t)

Le constructeur recopie les caractères de la string dans l'ensemble dont le type est donc **char**.

Convertit l'ensemble en tableau object[ ] (on a perdu le type).

```
public int Add( object value )  {
 // ajoute un élément à l'ensemble sinon -1 ou -2
 if (this.Count==0)
    FTypeElement = value.GetType( );
 if (value.GetType( ) ==TypeElement)  {
  if (!Contains(value) )
   return( List.Add( value ) );
  else  {
   MessageBox.Show("L'élément "+value.ToString()+", est déjà présent !",
                   "Ensemble : insertion impossible (-1)");
   return -1;
  }
 }
 else  {
  MessageBox.Show("L'élément "+value.ToString()+", n'est pas du même type !",
                   "Ensemble : insertion impossible (-2)");
  return -2;
 }
}

public void Remove( object value )  {
 // enlève un élément à l'ensemble s'il est présent et de même type
 if (value.GetType( )==FTypeElement && Contains(value))
  List.Remove( value );
}

public bool Contains( object value )  {
 // true si value est dans type setOfChar, false sinon.
 return( List.Contains( value ) );
}

protected override void OnInsert( int index, Object value )  {
 Inserer( this , EventArgs.Empty );
}

protected override void OnRemove( int index, Object value )  {
 Enlever( this , EventArgs.Empty );
}
}
```

Extrait de code de test où E1, E2 et EnsCar sont des setOfObject

```
//-- chargement par ajout élément par élément :
E1.Add('a');
E1.Add('b');
E1.Add('c');
E1.Add('d');
E1.Add('e');
//-- chargement par string :
E2 = new setOfObject("xyztu#");
EnsCar = E1 + E2 ;
foreach (object elt in EnsCar)
  Console.WriteLine("    "+ elt );

Console.WriteLine("type d'élément de
l'ensemble : "+EnsCar.TypeElement);
```

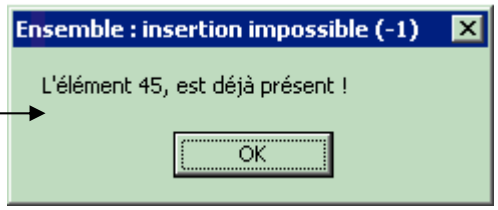
On teste sur le type d'élément **char**.

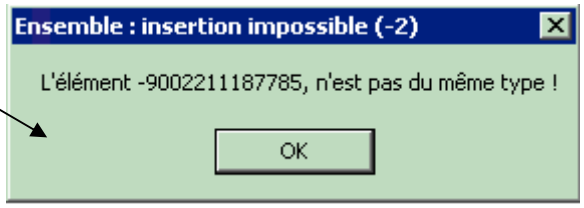
```
Contenu de l'ensemble EnsCar :
    a
    b
    c
    d
    e
    x
    y
    z
    t
    u
    #
type d'élément de l'ensemble : System.Char
```

Extrait de code de test où E1, E2 et EnsCar sont des setOfObject

```
//-- chargement par ajout élément par élément :
E1.Add(45);
E1.Add(-122);
E1.Add(666);

E1.Add(45);

E1.Add(-9002211187785);

E1.Add(12);

//-- chargement par ajout d'un tableau d'entiers :
int[ ] table=new int[ ]{ 125,126,127};
E2 = new setOfObject(table);
EnsCar = E1 + E2 ;

foreach (object elt in EnsCar)
  Console.WriteLine("    "+ elt );

Console.WriteLine("type d'élément de l'ensemble : "+EnsCar.TypeElement);
```

On teste sur le type d'élément **int**.





-9002211187785 est de type **long** et non **int** !

```
Contenu de l'ensemble EnsCar :
    45
    -122
    666
    12
    125
    126
    127
type d'élément de l'ensemble : System.Int32
```

classe C# de tests de la classe *setOfObject* : code source

```
class UtilisesetOfObject{

 private static void EnsbleOnEnlever( object sender,  EventArgs e )   {
   Console.WriteLine("On va enlever un élément de l'ensemble");
 }

 private static void version (int choix) {
  // Déclare les ensembles
  setOfObject Ensble1 = new setOfObject();
  setOfObject EnsCar,Ensble2;

  // Initialise les ensembles selon le paramètre
  switch(choix)  {
   case 1: ChargerEnsChar(Ensble1,out Ensble2);   break;
   case 2: ChargerEnsInt(Ensble1,out Ensble2);   break;
   case 3: ChargerEnsTChar(Ensble1,out Ensble2);   break;
   case 4: ChargerEnsArrayList(Ensble1,out Ensble2);   break;
   case 5: ChargerEnsArrayString(Ensble1,out Ensble2);   break;
   case 6: ChargerJours(Ensble1,out Ensble2);   break;
   default:ChargerEnsChar(Ensble1,out Ensble2);   break;
  }
  Console.WriteLine("card(Ensble1)="+Ensble1.Card+" ; ");
  Console.WriteLine("card(Ensble2)="+Ensble2.Card+" ; ");
```

Test d'utilisation de différents type d'éléments dans l'ensemble.

```
 EnsCar=Ensble1+Ensble2;
 EnsCar.OnEnlever += new EventHandler(EnsbleOnEnlever);
 Console.WriteLine("card(EnsCar)="+EnsCar.Card+" ; ");
 Console.WriteLine();

 // Affichage du Contenu de l'ensemble avec for :
 Console.WriteLine( "Contenu de l'ensemble EnsCar :" );
 Console.WriteLine( "========== for ============" );
 for(int i=0; i<EnsCar.Card; i++)
  Console.WriteLine("    "+ EnsCar[i] );

 // Affichage du Contenu de l'ensemble avec foreach :
 Console.WriteLine( "========= foreach ==========" );
 foreach(object elt in EnsCar)
  Console.WriteLine("    "+ elt );

 // Affichage du type d'élément de l'ensemble
 Console.WriteLine("type d'élément de l'ensemble : "+EnsCar.TypeElement);
}

private static void ChargerEnsInt( setOfObject E1,out setOfObject E2) {
 //-- chargement par ajout élément par élément :
 E1.Add(45);
 E1.Add(-122);
 E1.Add(666);
 E1.Add(45);
 E1.Add(-9002211187785);
 E1.Add(12);
 //-- chargement par ajout d'un tableau d'entiers :
 int[ ] table=new int[ ] {  125,126,127  };
 E2 = new setOfObject(table);
}

private static void ChargerEnsChar( setOfObject E1,out setOfObject E2) {
 //-- chargement par ajout élément par élément :
 E1.Add('a');
 E1.Add('b');
 E1.Add('c');
 E1.Add('d');
 E1.Add('e');
 //-- chargement par string :
 E2 = new setOfObject("xyztu#");
}

private static void ChargerEnsTChar( setOfObject E1,out setOfObject E2) {
 //-- chargement par ajout élément par élément :
 E1.Add('a');
 E1.Add('b');
 E1.Add('c');
 E1.Add('d');
 E1.Add('e');
 //-- chargement par tableau de char :
 char[ ] table = new char [ ] {  'x','y','z','t','u','#'};
  E2=new setOfObject(table);
 }

 private static void ChargerEnsArrayList( setOfObject E1,out setOfObject E2)  {
  //-- chargement par ajout élément par élément :
  E1.Add("un");
  E1.Add("deux");
```

Appel au constructeur :

public setOfObject(Array t)

Appel au constructeur :

public setOfObject(string s)

Appel au constructeur :

public setOfObject(Array t)

```
 E1.Add("trois");
 E1.Add("quatre");
 E1.Add("cinq");
 //-- chargement par ArrayList de string :
 ArrayList t = new ArrayList( );
 t.Add("six");
 t.Add("sept");
 t.Add("fin.");
 E2=new setOfObject(t);
}

private static void ChargerEnsArrayString( setOfObject E1,out setOfObject E2)  {
 //-- chargement par ajout élément par élément :
 E1.Add("rat");
 E1.Add("chien");
 E1.Add("chat");
 E1.Add("vache");
 //-- chargement par tableau de string :
 string[ ] table = new  string[3];
 table[0]="voiture";
 table[1]="bus";
 table[2]="fin.";
 E2=new setOfObject(table);
}
//--un type énuméré
enum Jours   {   Dimanche, Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi  }

private static void ChargerJours( setOfObject E1,out setOfObject E2)  {
 //-- chargement par ajout élément par élément :
 E1.Add(Jours.Vendredi);
 E1.Add(Jours.Samedi);
 E1.Add(Jours.Dimanche);
 //-- chargement par tableau de TypePerso :
 Jours[ ]table = new Jours[4];
 table[0]=Jours.Lundi;
 table[1]=Jours.Mardi;
 table[2]=Jours.Mercredi;
 table[3]=Jours.Jeudi;
 E2=new setOfObject(table);
}

[STAThread]
public static void Main( )   {
 Console.WriteLine("------------ 1 -----------------");
 version(1);
 Console.WriteLine("----------- 2 ------------------");
 version(2);
 Console.WriteLine("----------- 3 ------------------");
 version(3);
 Console.WriteLine("----------- 4 ------------------");
 version(4);
 Console.WriteLine("----------- 5 ------------------");
 version(5);
 Console.WriteLine("----------- 6 ------------------");
 version(6);
 }
}
```

Appel au constructeur :
public setOfObject(ArrayList t) : **this** (t.ToArray( ))

qui appelle lui-même :
public setOfObject(object[ ] t)

Appel au constructeur :

public setOfObject(object[ ] t)

Appel au constructeur :

public setOfObject(Array t)

Exécutez le programme et observez le type de l'ensemble dans chaque cas.

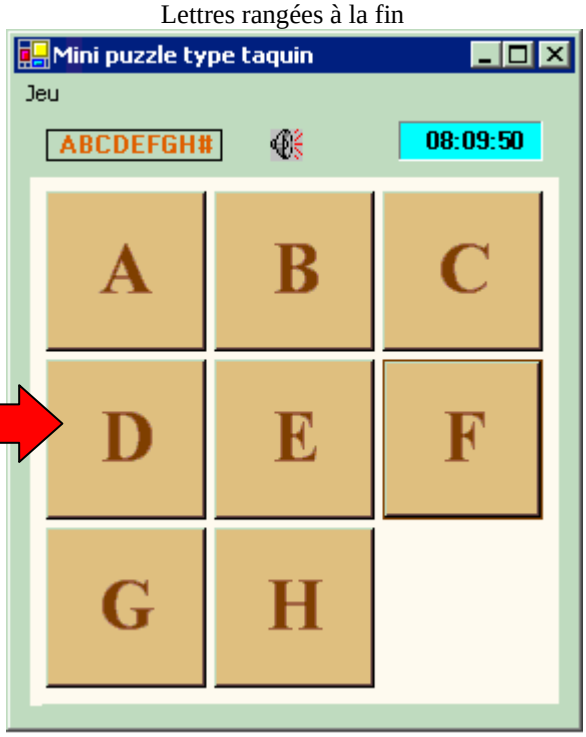
Classes, objet et IHM

Construction d'un jeu : puzzle genre "taquin"

Soit à construire une interface de jeu du taquin permettant de ranger par ordre alphabétique, des lettres disposées dans n'importe quel ordre sur un damier 3 x 3 :

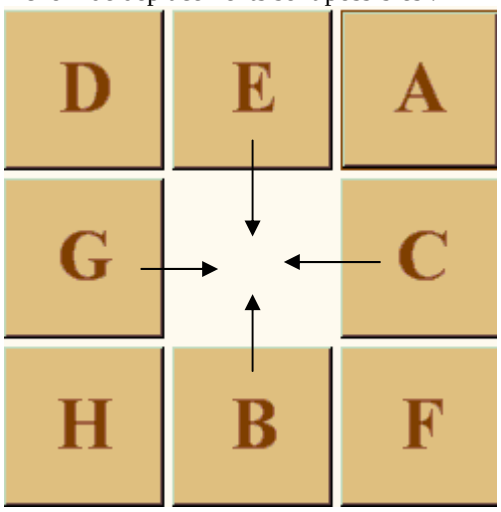
Lettres au départ dans le désordre

Lettres rangées à la fin

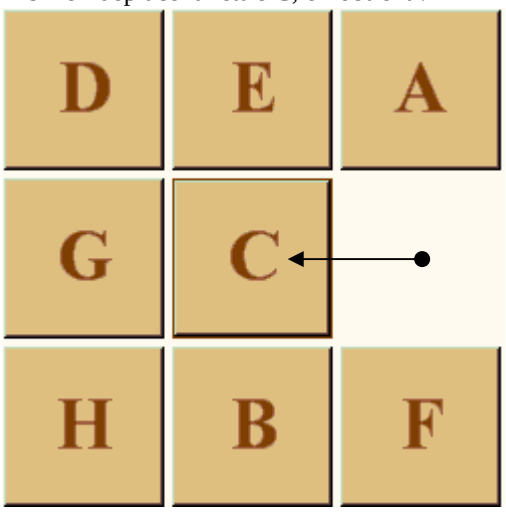


Sur les 9 cases une seule est disponible, le jeu consiste à ne déplacer qu'une seule lettre à la fois et uniquement dans la case restée libre. Par exemple dans la configuration de départ ci-haut seules les lettres G, E, C , B peuvent être déplacées vers la case centrale restée libre.

4 choix de déplacements sont possibles :

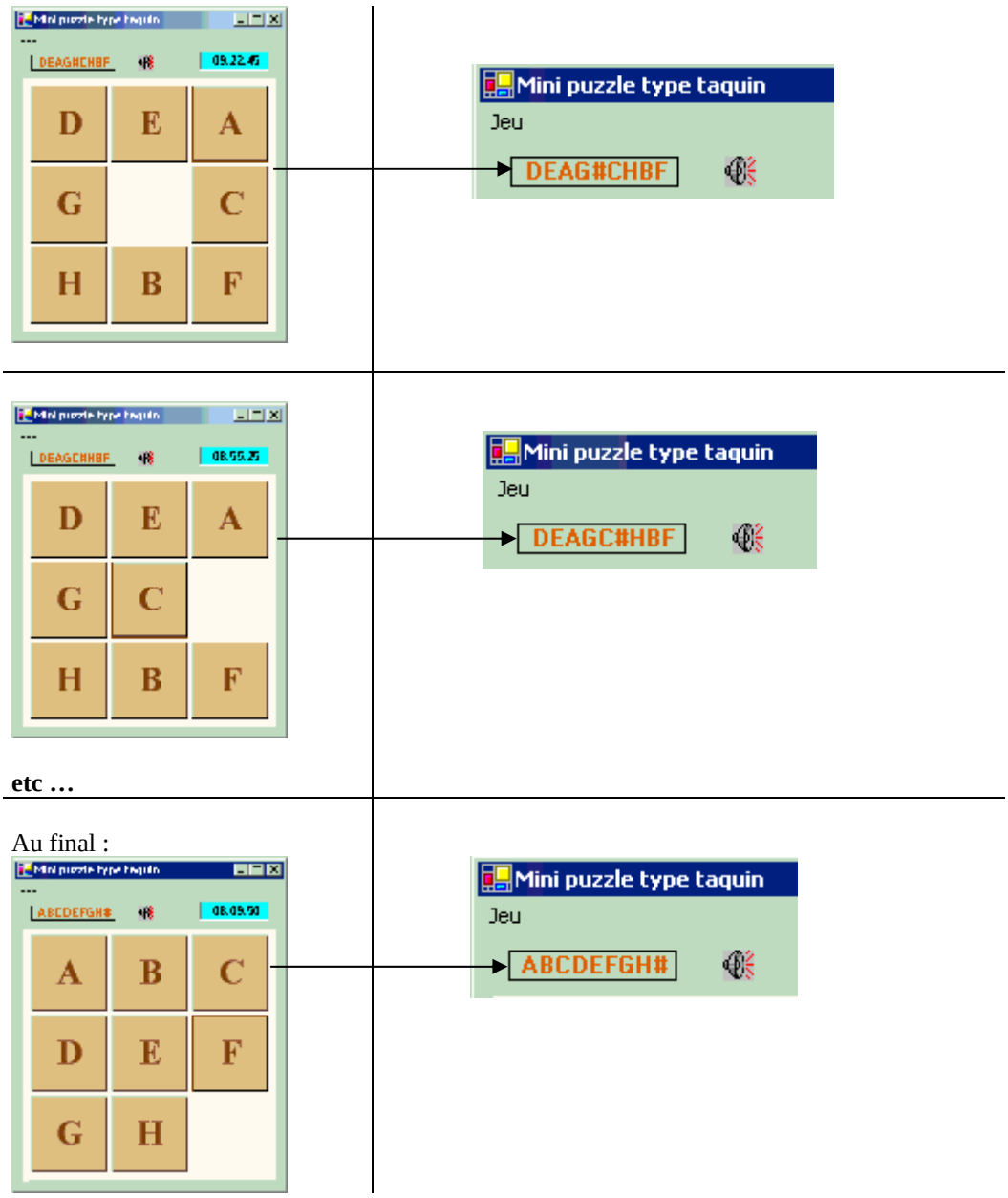


si l'on déplace la lettre C, on obtient :



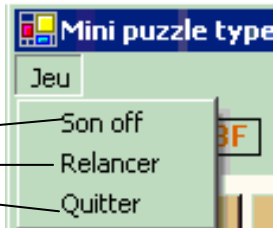
L'interface à construire en C# doit permettre à un utilisateur de jouer au taquin dans les conditions de contraintes du jeu, il utilisera la souris pour cliquer sur une lettre afin de la déplacer vers la case libre (le programme doit gérer la possibilité pour une case d'être déplacée ou non et doit tester à chaque déplacement si le joueur a gagné ou non).

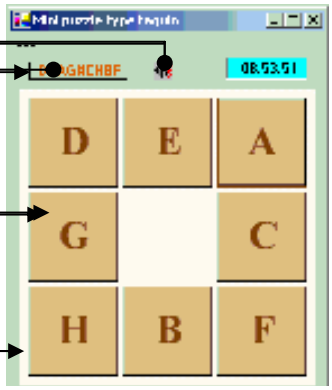
L'IHM affichera la liste des lettres lue depuis le coin supérieur gauche jusqu'au coin inférieur droit du tableau (la case libre sera représentée par un #)



Lorsque le joueur a trouvé la bonne combinaison et rangé les lettres dans le bon ordre, prévoir de lui envoyer une fenêtre de dialogue de félicitation et un menu lui permettant de rejouer ou de quitter le jeu:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Runtime.InteropServices;


namespace WinAppliPuzzle
{
 /// <summary>
 /// Description résumée de Form1.
 /// </summary>
 public class Form1 : System.Windows.Forms.Form {
  private System.Windows.Forms.MainMenu mainMenu1;
  private System.Windows.Forms.Label labelHorloge;
  private System.Windows.Forms.MenuItem menuItemjeu;
  private System.Windows.Forms.MenuItem menuItemson;
  private System.Windows.Forms.MenuItem menuItemrelancer;
  private System.Windows.Forms.MenuItem menuItemquitter;
  private System.Windows.Forms.Timer timerTemps;

  private System.ComponentModel.IContainer components;
  private const String correct="ABCDEFGH#";
  private String modele=correct;
  private int trou=9;
  private Point Ptrou=new Point(176,176);
  private System.Windows.Forms.PictureBox pictureBoxSon;
  private System.Windows.Forms.Label labelModele;
  private System.Windows.Forms.Button buttonA;
  private System.Windows.Forms.Button buttonB;
  private System.Windows.Forms.Button buttonC;
  private System.Windows.Forms.Button buttonF;
  private System.Windows.Forms.Button buttonE;
  private System.Windows.Forms.Button buttonD;
  private System.Windows.Forms.Button buttonH;
  private System.Windows.Forms.Button buttonG;
  private System.Windows.Forms.Panel panelFond;
  private bool Sonok=false;

  private int TestDeplace(int num)  {
   switch (num)    {
    case 1:if(trou==2 | trou==4) return trou;    break;
    case 2:if(trou==1 | trou==3 | trou==5) return trou;    break;
    case 3:if(trou==2 | trou==6) return trou;    break;
    case 4:if(trou==1 | trou==5 | trou==7) return trou;    break;
    case 5:if(trou==2 | trou==4 | trou==6 | trou==8) return trou;    break;
    case 6:if(trou==3 | trou==5 | trou==9) return trou;    break;
    case 7:if(trou==4 | trou==8) return trou;    break;
    case 8:if(trou==5 | trou==7 | trou==9) return trou;    break;
    case 9:if(trou==6 | trou==8) return trou;    break;
   }
   return -1;
  }

  [DllImport("user32.dll",EntryPoint="MessageBeep")]
  public static extern bool MessageBeep(uint uType);
```





Un exemple d'utilisation de fonction non CLS compatible et dépendant de la plateforme Win32. (envoyer un son sur le haut-parleur du PC) .

```
private void DeplaceVers(Button source, int but)  {
 int T,L;
 if(but>0)   {
  T = source.Location.Y;
  L = source.Location.X;
  source.Location = Ptrou;
  Ptrou = new Point(L,T);
  if (Sonok)
   MessageBeep(0);
 }
}

private void MajModele(Button source, int but)  {
 if(but>0)   {
  char[ ] s = modele.ToCharArray();
  s[(int)(source.Tag)-1]='#';
  s[but-1]=(char)(source.Text[0]);
  trou = (int)(source.Tag);
  source.Tag=but;
  modele="";
  for(int i=0; i<s.Length;i++)
   modele = modele+s[i];
 }
}

public void Tirage( )  {
 buttons_Click(buttonF,  new EventArgs( ));
 buttons_Click(buttonE,  new EventArgs( ));
 buttons_Click(buttonB,  new EventArgs( ));
 buttons_Click(buttonA,  new EventArgs( ));
 buttons_Click(buttonD,  new EventArgs( ));
 buttons_Click(buttonG,  new EventArgs( ));
 buttons_Click(buttonH,  new EventArgs( ));
 buttons_Click(buttonB,  new EventArgs( ));
 buttons_Click(buttonE,  new EventArgs( ));
 buttons_Click(buttonC,  new EventArgs( ));
 buttons_Click(buttonA,  new EventArgs( ));
 buttons_Click(buttonE,  new EventArgs( ));
}
```

Ecrivez une autre redisposition aléatoire des boutons. Ici nous lançons une séquence modifiant l'existant en utilisant les gestionnaires de click de souris comme si l'utilisateur avait cliqué 12 fois sur l'IHM.

```
public Form1( )  {
 //
 // Requis pour la prise en charge du Concepteur Windows Forms
 //
 InitializeComponent( );

 //
 // TODO : ajoutez le code du constructeur après l'appel à InitializeComponent
 //
}

/// <summary>
/// Nettoyage des ressources utilisées.
/// </summary>
protected override void Dispose( bool disposing )  {
 if( disposing )   {
  if (components != null)    {
   components.Dispose( );
  }
 }
```

```
 base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Méthode requise pour la prise en charge du concepteur - ne modifiez pas
/// le contenu de cette méthode avec l'éditeur de code.
/// </summary>
private void InitializeComponent()
{
 this.components = new System.ComponentModel.Container( );
 System.Resources.ResourceManager resources = new System.Resources.ResourceManager(typeof(Form1));
 this.mainMenu1 = new System.Windows.Forms.MainMenu( );
 this.menuItemjeu = new System.Windows.Forms.MenuItem( );
 this.menuItemson = new System.Windows.Forms.MenuItem( );
 this.menuItemrelancer = new System.Windows.Forms.MenuItem( );
 this.menuItemquitter = new System.Windows.Forms.MenuItem( );
 this.panelFond = new System.Windows.Forms.Panel( );
 this.buttonH = new System.Windows.Forms.Button( );
 this.buttonG = new System.Windows.Forms.Button( );
 this.buttonF = new System.Windows.Forms.Button( );
 this.buttonE = new System.Windows.Forms.Button( );
 this.buttonD = new System.Windows.Forms.Button( );
 this.buttonC = new System.Windows.Forms.Button( );
 this.buttonB = new System.Windows.Forms.Button( ) ;
 this.buttonA = new System.Windows.Forms.Button( ) ;
 this.pictureBoxSon = new System.Windows.Forms.PictureBox( );
 this.labelModele = new System.Windows.Forms.Label( );
 this.timerTemps = new System.Windows.Forms.Timer(this.components);
 this.labelHorloge = new System.Windows.Forms.Label( );
 this.panelFond.SuspendLayout( );
 this.SuspendLayout( );
 //
 // mainMenu1
 //
 this.mainMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[]
 {
  this.menuItemjeu
 }
 );
 //
 // menuItemjeu
 //
 this.menuItemjeu.Index = 0;
 this.menuItemjeu.MenuItems.AddRange(new System.Windows.Forms.MenuItem[]
 {
  this.menuItemson,
  this.menuItemrelancer,
  this.menuItemquitter
 }
 );
 this.menuItemjeu.Text = "Jeu";
 //
 // menuItemson
 //
 this.menuItemson.Index = 0;
 this.menuItemson.Text = "Son off";
 this.menuItemson.Click += new System.EventHandler(this.menuItemson_Click);
 //
 // menuItemrelancer
```

```
//
this.menuItemrelancer.Index = 1;
this.menuItemrelancer.Text = "Relancer";
this.menuItemrelancer.Click += new System.EventHandler(this.menuItemrelancer_Click);
//
// menuItemquitter
//
this.menuItemquitter.Index = 2;
this.menuItemquitter.Text = "Quitter";
this.menuItemquitter.Click += new System.EventHandler(this.menuItemquitter_Click);
//
// panelFond
//
this.panelFond.BackColor = System.Drawing.SystemColors.Info;
this.panelFond.Controls.Add(this.buttonH);
this.panelFond.Controls.Add(this.buttonG);
this.panelFond.Controls.Add(this.buttonF);
this.panelFond.Controls.Add(this.buttonE);
this.panelFond.Controls.Add(this.buttonD);
this.panelFond.Controls.Add(this.buttonC);
this.panelFond.Controls.Add(this.buttonB);
this.panelFond.Controls.Add(this.buttonA);
this.panelFond.ImeMode = System.Windows.Forms.ImeMode.NoControl;
this.panelFond.Location = new System.Drawing.Point(8, 32);
this.panelFond.Name = "panelFond";
this.panelFond.Size = new System.Drawing.Size(264, 264);
this.panelFond.TabIndex = 0;
//
// buttonH
//
this.buttonH.BackColor = System.Drawing.Color.Tan;
this.buttonH.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                        System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonH.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonH.Location = new System.Drawing.Point(92, 176);
this.buttonH.Name = "buttonH";
this.buttonH.Size = new System.Drawing.Size(80, 80);
this.buttonH.TabIndex = 16;
this.buttonH.Tag = 8;
this.buttonH.Text = "H";
this.buttonH.Click += new System.EventHandler(this.buttons_Click);
this.buttonH.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonG
//
this.buttonG.BackColor = System.Drawing.Color.Tan;
this.buttonG.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                       System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonG.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonG.Location = new System.Drawing.Point(8, 176);
this.buttonG.Name = "buttonG";
this.buttonG.Size = new System.Drawing.Size(80, 80);
this.buttonG.TabIndex = 15;
this.buttonG.Tag = 7;
this.buttonG.Text = "G";
this.buttonG.Click += new System.EventHandler(this.buttons_Click);
this.buttonG.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonF
//
```

```
this.buttonF.BackColor = System.Drawing.Color.Tan;
this.buttonF.Cursor = System.Windows.Forms.Cursors.Default;
this.buttonF.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                        System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonF.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonF.Location = new System.Drawing.Point(176, 92);
this.buttonF.Name = "buttonF";
this.buttonF.Size = new System.Drawing.Size(80, 80);
this.buttonF.TabIndex = 14;
this.buttonF.Tag = 6;
this.buttonF.Text = "F";
this.buttonF.Click += new System.EventHandler(this.buttons_Click);
this.buttonF.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonE
//
this.buttonE.BackColor = System.Drawing.Color.Tan;
this.buttonE.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                       System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonE.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonE.Location = new System.Drawing.Point(92, 92);
this.buttonE.Name = "buttonE";
this.buttonE.Size = new System.Drawing.Size(80, 80);
this.buttonE.TabIndex = 13;
this.buttonE.Tag = 5;
this.buttonE.Text = "E";
this.buttonE.Click += new System.EventHandler(this.buttons_Click);
this.buttonE.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonD
//
this.buttonD.BackColor = System.Drawing.Color.Tan;
this.buttonD.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                        System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonD.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonD.Location = new System.Drawing.Point(8, 92);
this.buttonD.Name = "buttonD";
this.buttonD.Size = new System.Drawing.Size(80, 80);
this.buttonD.TabIndex = 12;
this.buttonD.Tag = 4;
this.buttonD.Text = "D";
this.buttonD.Click += new System.EventHandler(this.buttons_Click);
this.buttonD.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonC
//
this.buttonC.BackColor = System.Drawing.Color.Tan;
this.buttonC.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                        System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonC.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonC.Location = new System.Drawing.Point(176, 8);
this.buttonC.Name = "buttonC";
this.buttonC.Size = new System.Drawing.Size(80, 80);
this.buttonC.TabIndex = 11;
this.buttonC.Tag = 3;
this.buttonC.Text = "C";
this.buttonC.Click += new System.EventHandler(this.buttons_Click);
this.buttonC.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonB
```

```
//
this.buttonB.BackColor = System.Drawing.Color.Tan;
this.buttonB.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                    System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonB.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonB.Location = new System.Drawing.Point(92, 8);
this.buttonB.Name = "buttonB";
this.buttonB.Size = new System.Drawing.Size(80, 80);
this.buttonB.TabIndex = 10;
this.buttonB.Tag = 2;
this.buttonB.Text = "B";
this.buttonB.Click += new System.EventHandler(this.buttons_Click);
this.buttonB.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// buttonA
//
this.buttonA.BackColor = System.Drawing.Color.Tan;
this.buttonA.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                    System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonA.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonA.Location = new System.Drawing.Point(8, 8);
this.buttonA.Name = "buttonA";
this.buttonA.Size = new System.Drawing.Size(80, 80);
this.buttonA.TabIndex = 9;
this.buttonA.Tag = 1;
this.buttonA.Text = "A";
this.buttonA.Click += new System.EventHandler(this.buttons_Click);
this.buttonA.MouseMove += new System.Windows.Forms.MouseEventHandler(this.buttons_MouseMove);
//
// pictureBoxSon
//
this.pictureBoxSon.Image = ((System.Drawing.Image)(resources.GetObject("pictureBoxSon.Image")));
this.pictureBoxSon.Location = new System.Drawing.Point(128, 8);
this.pictureBoxSon.Name = "pictureBoxSon";
this.pictureBoxSon.Size = new System.Drawing.Size(16, 16);
this.pictureBoxSon.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize;
this.pictureBoxSon.TabIndex = 1;
this.pictureBoxSon.TabStop = false;
//
// labelModele
//
this.labelModele.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
this.labelModele.Font = new System.Drawing.Font("Microsoft Sans Serif", 8.25F,
                        System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.labelModele.ForeColor = System.Drawing.Color.Chocolate;
this.labelModele.Location = new System.Drawing.Point(16, 8);
this.labelModele.Name = "labelModele";
this.labelModele.Size = new System.Drawing.Size(88, 16);
this.labelModele.TabIndex = 2;
this.labelModele.Text = "ABCDEFGH#";
this.labelModele.TextAlign = System.Drawing.ContentAlignment.TopCenter;
//
// timerTemps
//
this.timerTemps.Enabled = true;
this.timerTemps.Interval = 1000;
this.timerTemps.Tick += new System.EventHandler(this.timer1_Tick);
//
// labelHorloge
//
```

```
 this.labelHorloge.BackColor = System.Drawing.Color.Aqua;
 this.labelHorloge.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
 this.labelHorloge.Font = new System.Drawing.Font("Microsoft Sans Serif", 8.25F,
                                System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
 this.labelHorloge.Location = new System.Drawing.Point(192, 4);
 this.labelHorloge.Name = "labelHorloge";
 this.labelHorloge.Size = new System.Drawing.Size(72, 20);
 this.labelHorloge.TabIndex = 4;
 this.labelHorloge.Text = "00:00:00";
 this.labelHorloge.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
 //
 // Form1
 //
 this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
 this.ClientSize = new System.Drawing.Size(280, 305);
 this.Controls.Add(this.labelHorloge);
 this.Controls.Add(this.labelModele);
 this.Controls.Add(this.pictureBoxSon);
 this.Controls.Add(this.panelFond);
 this.Menu = this.mainMenu1;
 this.Name = "Form1";
 this.Text = "Mini puzzle type taquin";
 this.Load += new System.EventHandler(this.Form1_Load);
 this.panelFond.ResumeLayout(false);
 this.ResumeLayout(false);
}
#endregion

/// <summary>
/// Point d'entrée principal de l'application.
/// </summary>
[STAThread]
static void Main( )  {
 Application.Run(new Form1());

}

 private void buttons_Click(object sender, EventArgs e)  {
  int but = TestDeplace((int)((sender as Button).Tag) );
  DeplaceVers((sender as Button), but);
  MajModele((sender as Button),but);
  labelModele.Text = modele;
  if(modele.CompareTo(correct)==0)
   MessageBox.Show("Bravo vous avez trouvé !", "Information",
  MessageBoxButtons.OK, MessageBoxIcon.Information);
 }

private void timer1_Tick(object sender, System.EventArgs e)  {
  DateTime dt = DateTime.Now;
  labelHorloge.Text=dt.ToString("T");
 }

 private void buttons_MouseMove(object sender, System.Windows.Forms.MouseEventArgs e)  {
  if (TestDeplace((int)((sender as Button).Tag) )>0)
   (sender as Button).Cursor = System.Windows.Forms.Cursors.Hand;
  else
   (sender as Button).Cursor = System.Windows.Forms.Cursors.No;
 }

   //-- les gestionnaires des click dans le menu :
```

Gestionnaire centralisé du click **de souris sur les 8 boutons représentant les lettres dans le tableau.**

Gestionnaire centralisé du Move **de souris sur les 8 boutons représentant les lettres dans le tableau.**

```
 private void menuItemson_Click(object sender, System.EventArgs e)  {
  Console.WriteLine(menuItemson.Text);
  if(menuItemson.Text.IndexOf("off")>0)   {
   menuItemson.Text = "Son on";
   Sonok = false;
   pictureBoxSon.Visible = false;
  }
  else   {
   menuItemson.Text = "Son off";
   Sonok = true;
   pictureBoxSon.Visible = true;
  }
 }

 private void menuItemrelancer_Click(object sender, System.EventArgs e)  {
  Tirage( );
 }

 private void menuItemquitter_Click(object sender, System.EventArgs e)  {
  Close( );
 }

 private void Form1_Load(object sender, System.EventArgs e)  {
  Tirage( );
  Sonok = true;
 }
}

}
```

Remarques

Nous avons utilisé la classe **System.Windows.Forms.Timer** afin d'afficher l'heure systèmedans le jeu :

System.Object
|__ System.MarshalByRefObject
|__ System.ComponentModel.Component
|__ System.Windows.Forms.Timer

Cette classe très semblable à la classe TTimer de Delphi, implémente une minuterie déclenchant un événement **Tick** (événement OnTimer en Delphi)selon un intervalle défini par l'utilisateur (préconisation Microsoft : un Timer doit être utilisé dans une fenêtre). Ci-dessous le gestionnaire de l'événement **Tick** :

```
private void timer1_Tick(object sender, System.EventArgs e)  {
  DateTime dt = DateTime.Now;
  labelHorloge.Text=dt.ToString("T");
 }
```

Récupère la date du jour (sous la forme d'un entier)

Converti l'entier au format : Hh : mm : ss

System.Object
|__ System.ValueType
|__ System.DateTime

Les valeurs de date de type **DateTime** sont mesurées en unités de 100 nanosecondes et exprimées sous forme d'un entier long.

Remarque concernant l'instruction chargeant une image :

```
this.pictureBoxSon.Image = ((System.Drawing.Image)(resources.GetObject("pictureBoxSon.Image")));
```

L'image utilisée sous le nom "pictureBoxSon.Image" doit figurer dans un fichier de ressource (ou y être ajoutée) suivant la manière dont le projet est construit (projet vide, application windows ...). L'instruction précédente permet alors l'accés à l'image "pictureBoxSon.Image" par l'accès au fichier de ressource.

Bibliographie

Livres papier vendus par éditeur

Livres C# en français

J.Richter, programmez microsoft .Net Framework, microsof press-Dunod, Paris (2002)
R.Standefer, ASP Net web training auto-formation, OEM-Eyrolles, Paris (2002)
G.Leblanc, solutions développeur C# et .NET, Eyrolles, Paris (2002)
B.Bischof, Langages .Net guide des équivalences, Eyrolles, Paris (2002)
S.Gross, cook book C#, Micro application, Paris (2002)
C.Eberhardt, le langage C#, campus press, Paris (2002)
H.Berthet, Visual C# concepts et mise en oeuvre, Ed.ENI, Nantes (2002)
M.Williams, manuel de référence microsof visual C# .Net, microsof press-Dunod, Paris (2003)
Kit de formation développer des applications windows avec visual C# .Net, microsof press-Dunod, Paris (2003)
Kit de formation développer des applications web avec visual C# .Net, microsof press-Dunod, Paris (2003)
V.Billotte, M.Thevenet, Le langage C#, Ed. Micro-Application, Paris (2002)
J.Gabillaud, ADO.NET l'accès aux données, Ed.ENI, Nantes (2004)
M.de Champlain, G.Patrick, C# 2.0 guide pratique du développeur, Dunod, Paris (2005)
D.Marshall Visual C# 2005, manuel de référence, microsof press, Paris (2006)

Site de l'association des développeurs francophones contenant de nombreux cours et tutoriels en ligne gratuits pour C++, Delphi, Java, C#, UML etc… :
http://www.developpez.com