Les premiers pas dans .Net Framework avec le langage C#

Cours d'initiation à la programmation en C#

RM di Scala *: révision du – 05 Septembre 2004*

426 pages

SOMMAIRE

Types, opérateurs, instructions

Introduction …….…. ……. P.3

Les outils élémentaires ……..…… P.4

Les éléments de base ……...……. P.12

Les opérateurs + exemples ……. P.22

Les instructions …….……. P.36

Les conditions ……. P.41

Les itérations ……. ……. P.47

Les ruptures de séquence …….. ……. P.51

Classes avec méthodes static …….……. …… P.55

Structures de données de base

Classe String ……..…. P.72

Tableaux, matrices ……..…. P.80

Collections, piles, files, listes …….……..…. P.92

C# est orienté objet

Classes, objets et méthodes ……..…. P.102

Polymorphisme d'objet …….……..…. P.136

Polymorphisme de méthode ……..…..……..…. P.148

Polymorphisme d'interfaces ……..…. P.172

Classe de délégation …….……..…. P.196

IHM avec C#

Les événements ……. **P.213**

Propriétés et indexeurs …….……. P.232

Fenêtres et ressources mémoires ……..……. P.257

Contrôles dans les formulaires ……..……. P.293

Exceptions comparées à Delphi et java ……..……. P.317

Exercices ……..……. P.327

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 C#Builder version personnelle, ou aller sur le site de Microsoft afin de connaître sa politique de téléchargement gratuits pour les outils .Net.

Remerciements : (*pour les corrections d'erreurs*)

A Vecchio56@free.fr, internaute averti sur la syntaxe de base C++ commune à Java et à C#.

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 comme la seule personne en dehors de moi qui a eu la constance de relire entièrement toutes les pages de l'ouvrage, alors que l'informatique n'est pas sa tasse de thé.

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 un site miroir du précédent et qui recommande le package pédagogique (http://rmdiscala.developpez.com/cours/) à ses visiteurs débutants.

Cette édition a été corrigée durant 2 mois de l'été 2004, elle a été optimisée en nombre de pages papier imprimables.

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 : discala@univ-tours.fr**

Introduction à

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 appplication, 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 porte la dénomination de **.NETFramework**, il est distribué gratuitement par microsoft sur toutes les versions de Windows (98, Me,..., Xp,...).

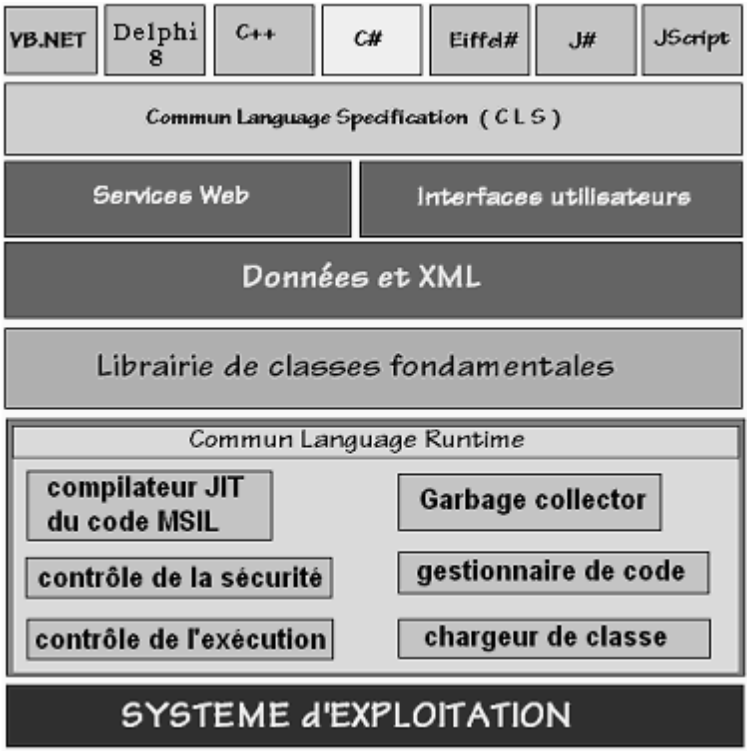
L'outil Visual Studio **.NET** contient l'environnement RAD de développement pour l'architecture **.NET**. Visual Studio **.NET** permet le développement d'applications classiques Windows ou Internet.

Dans ce document nous comparons souvent C# à ses deux parents Java et Delphi afin d'en signaler les apports et surtout les différences.

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 specifications communes 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**.

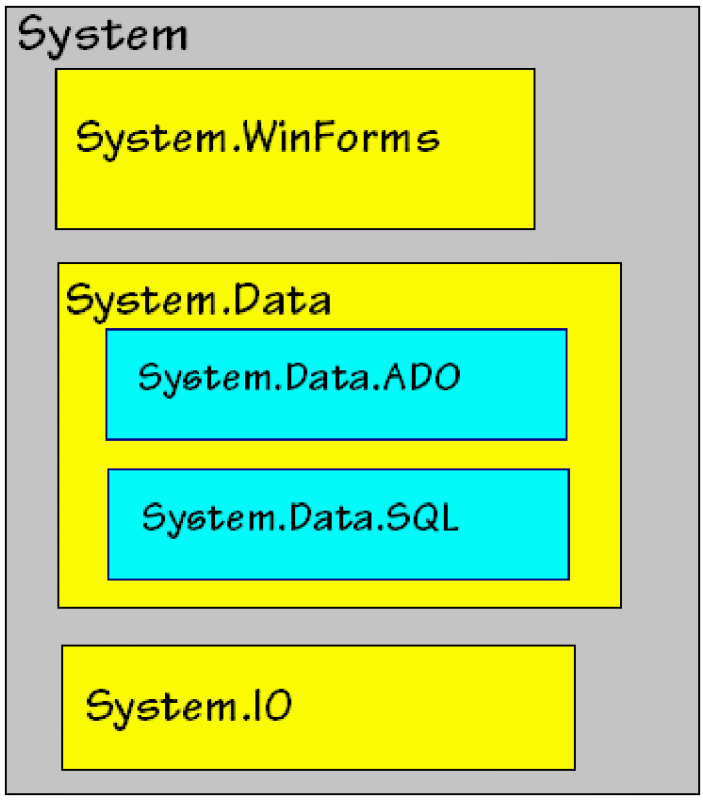
La seconde couche est un ensemble de composants graphiques disponibles dans Visual Studio **.NET** qui permettent de construire des interfaces homme-machine orientées Web (services Web) ou bien orientées applications classiques avec IHM.

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.

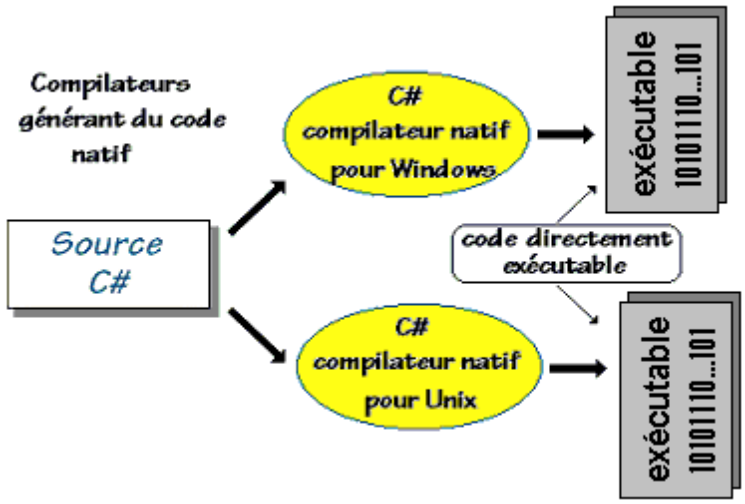
Donc dans le cas d'une p-machine le programme source est compilé, mais le programme cible est exécuté par l'interpréteur de la p-machine.

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

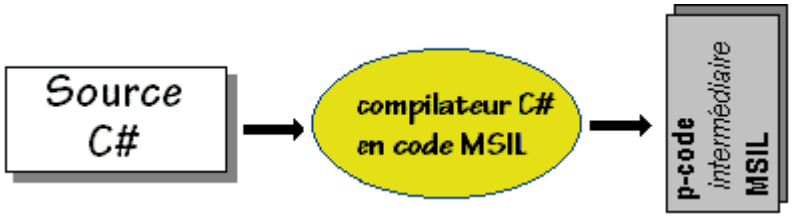
La compilation en bytecode (ou pseudo-code ou p-code ou code intermédiaire) est semblable à l'idée du p-code de N.Wirth pour obtenir un portage multi plate-formes du pascal. Le compilateur C# de **.NET Framework** traduit le programme source xxx.**cs** en un code intermédiaire indépendant de toute machine 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***

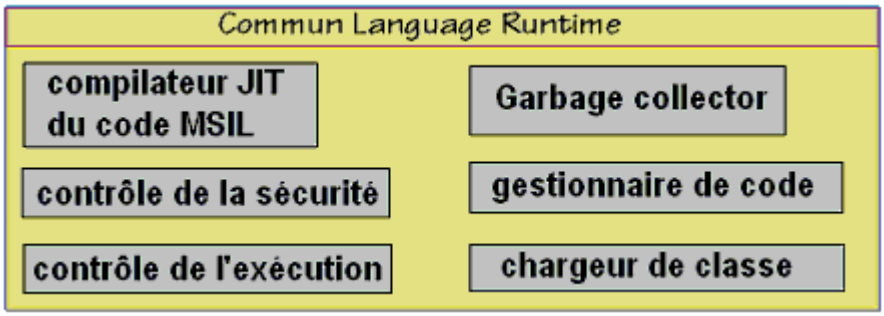


Programe source C# : Exemple.**cs**
Programe exécutable sous **.NET** : Exemple.**exe** (code portable IL)

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 sécutrité 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.

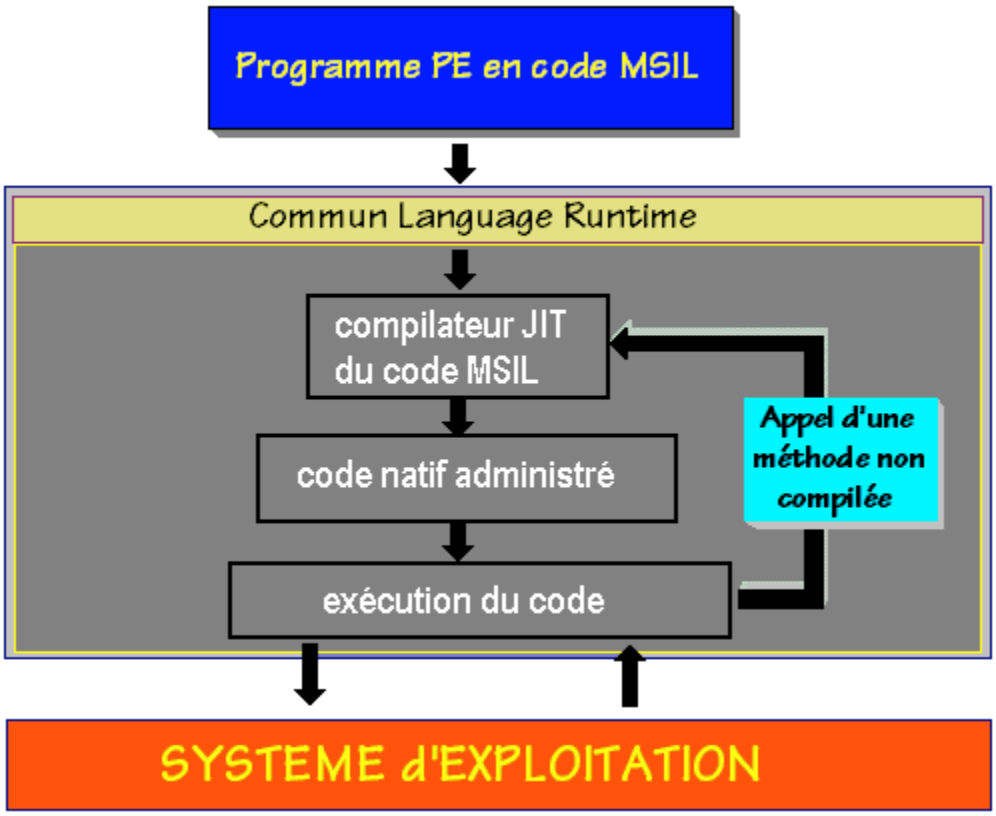
La compilation JIT progressive

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 :


Programme PE en code MSIL
Commun Language Runtime
compilateur JIT
du code MSIL
code natif administré
exécution du code
Appel d'une
méthode non
compilée
SYSTEME d'EXPLOITATION

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).

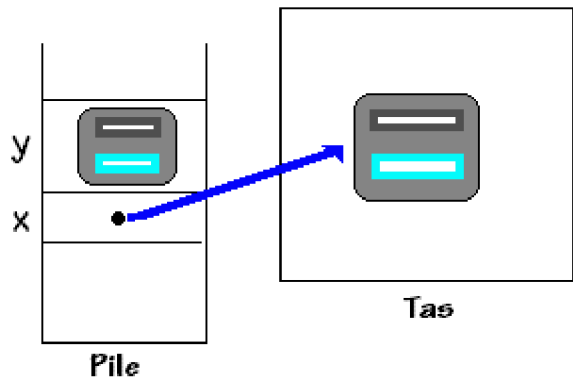
1. Les types valeurs du CLS dans .NET Framework

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 tout 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 65536)	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** qui 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.

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.

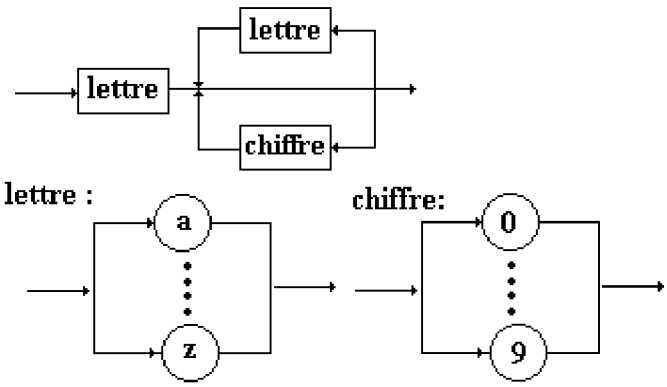
- Tous les types élémentaires peuvent être transtypés à l'exception du type **boolean** qui ne peut pas être converti en un autre type (différence avec le C).
- Les conversions **peuvent être restrictives** quant au résultat; par exemple le transtypage du réel 5.27e-2 en entier (x = **(int)**5.27e-2)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

∀p ∈ { **V**, **F** } , ∀q ∈ { **V**, **F** } , p **&**q est toujours évalué en entier *(p et q sont toujours évalués).*
∀p ∈ { **V**, **F** } , ∀q ∈ { **V**, **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
∀q ∈ { **V**, **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 ou noté **&&** qui a **la même table de vérité** que l'opérateur **&** mais qui applique ce théorème.

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

L'opérateur ou optimisé : | |

Théorème
∀q ∈ { **V**, **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

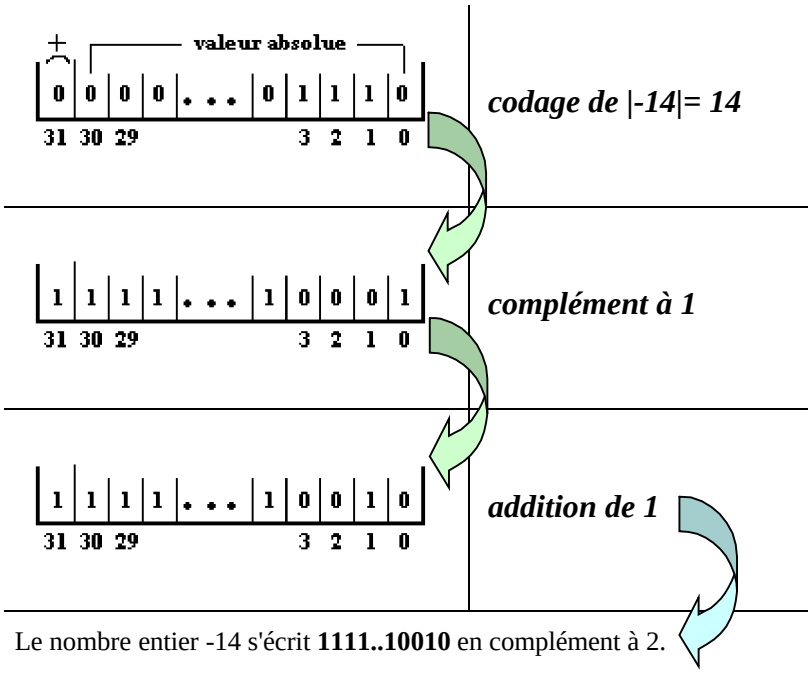
Afin de bien comprendre ces opérateurs, le lecteur doit bien connaître les différents codages des entiers en machine (binaire pur, binaire signé, complément à deux) car les entiers C# sont codés en complément à deux et la manipulation bit à bit nécessite une bonne compréhension de ce codage.

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.

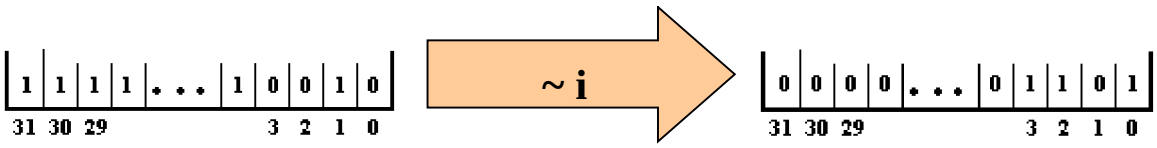
Soient la déclaration C# suivante :

int i = -14 , j ;

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

- Etude de l'instruction : j = ~ i

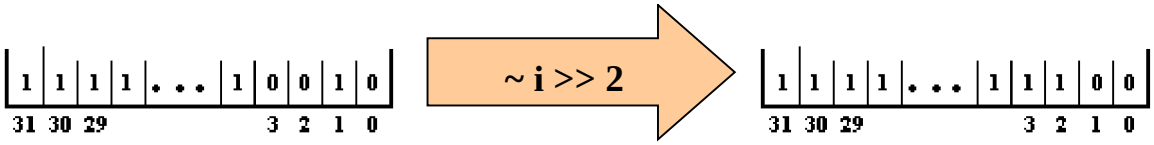
j = ~ i ; *// complémentation des bits de i*



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

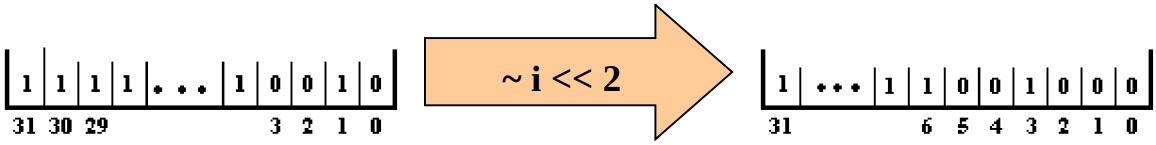
j = i >> 2 ; *// décalage avec signe de 2 bits vers la droite*



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

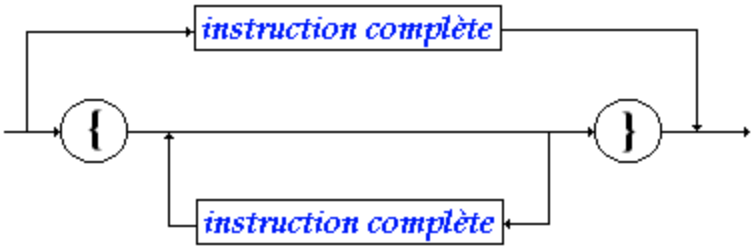
1 - les instructions de bloc

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 :

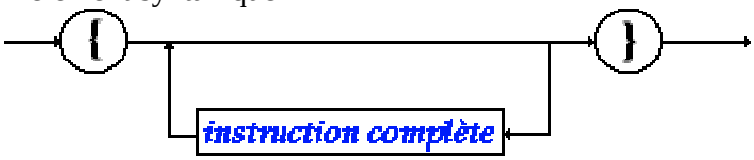


instruction complète :

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 ;
    }
  }
}
```

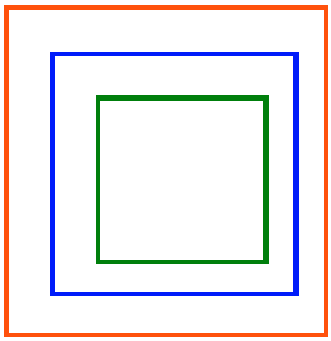


schéma d'imbrication des 3 blocs

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

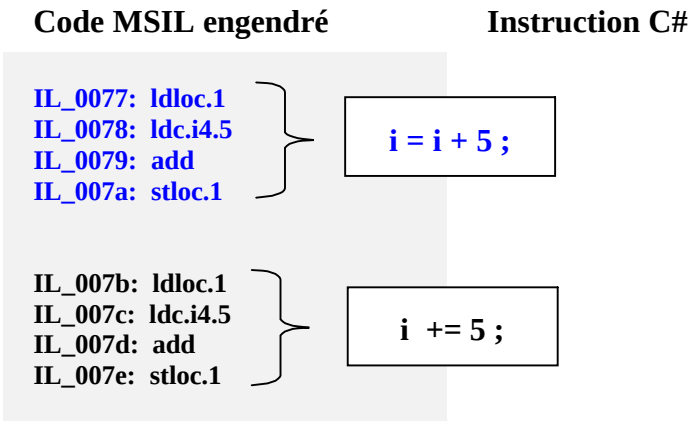
{ +, - , * , / , % , << , >> , >>> , & , | , ^ },

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 :



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
```

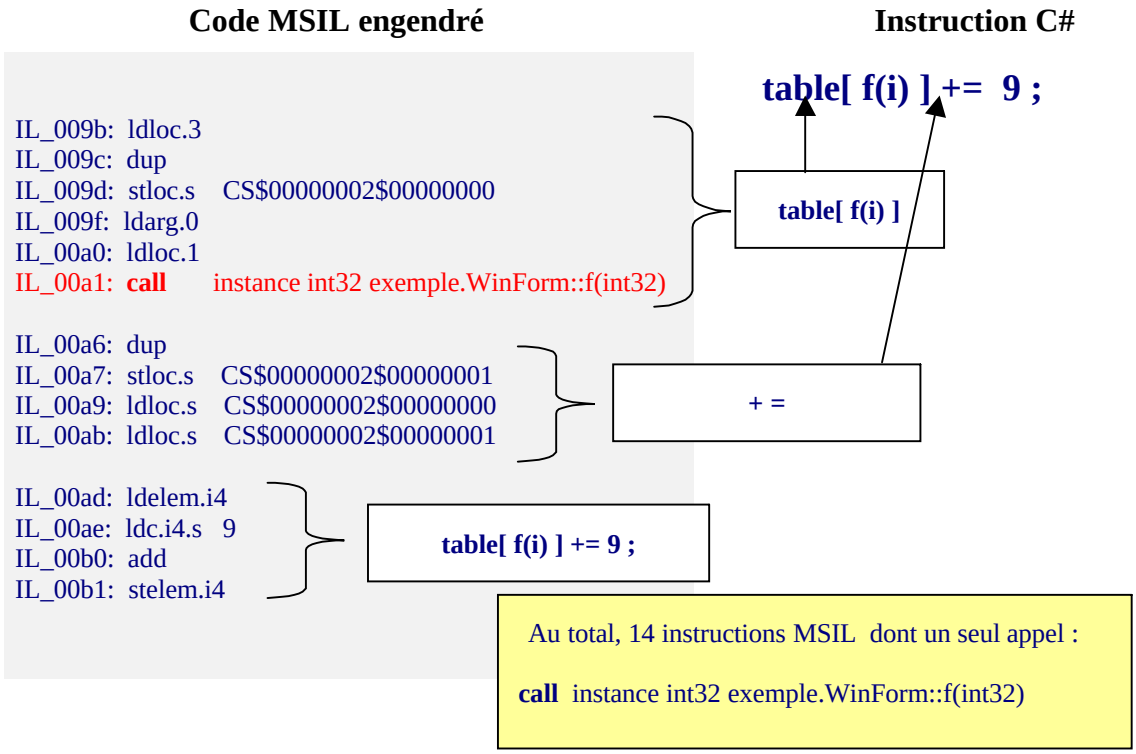
table[f(i)]

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

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

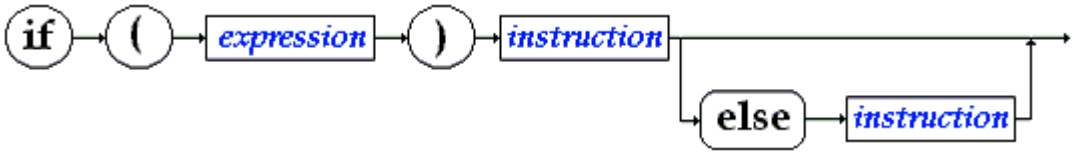
Info MSIL :

> **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 :

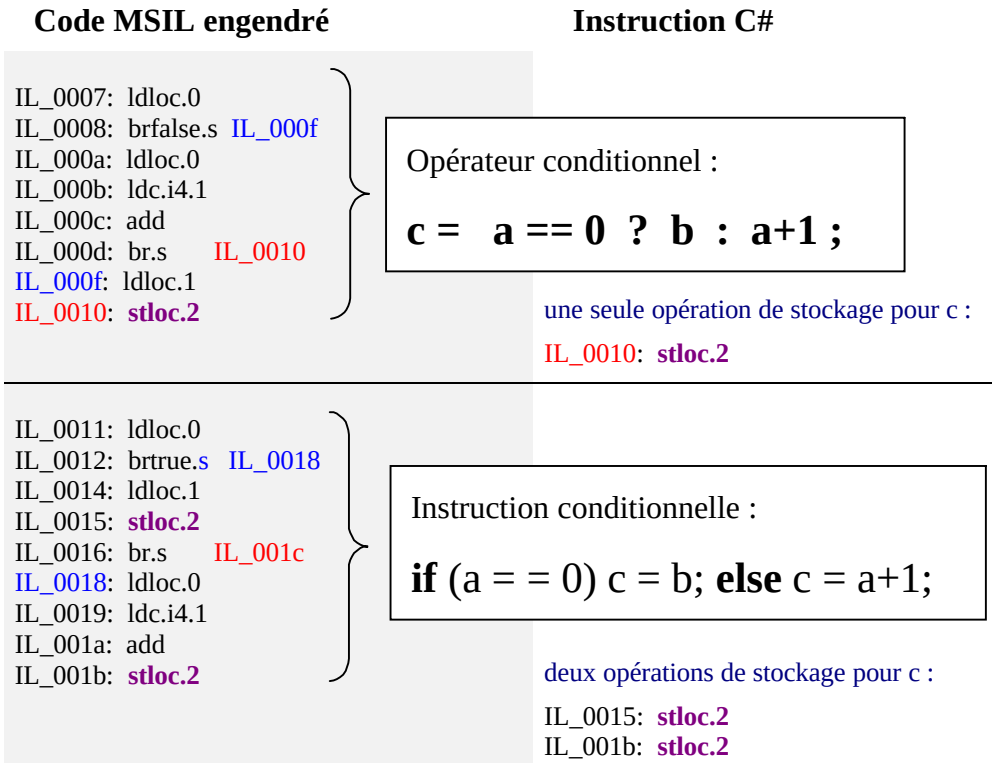
Exemple :

```
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;
```



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) ;

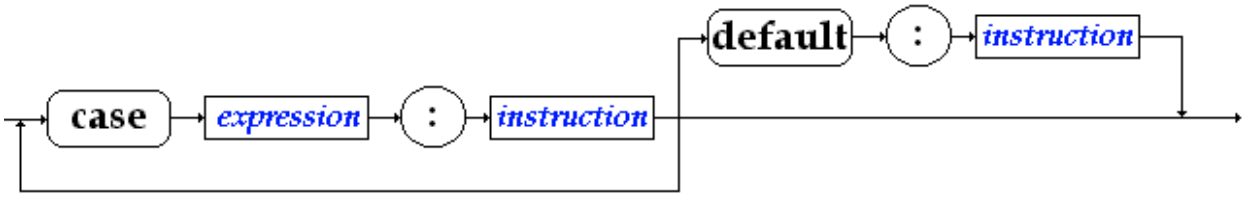
> ***réponse :***
> *a,b,c contiennent après exécution le plus grand des deux entiers contenus au départ dans a et b.*

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** ou bien **short**.
- La partie expression d'un bloc switch doit être une constante ou une valeur immédiate du type **byte**, **char**, **int** ou bien **short**.
- **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 égal. 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).

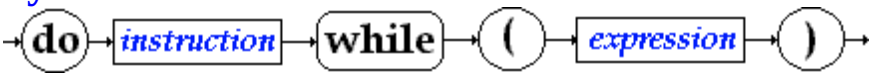
Sémantique :
Identique à celle du pascal (instruction algorithmique **tantque** .. **faire** .. **ftant**) avec le même défaut de fermeture de la boucle.

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 **répéter** Instr **jusquà non** Expr.

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

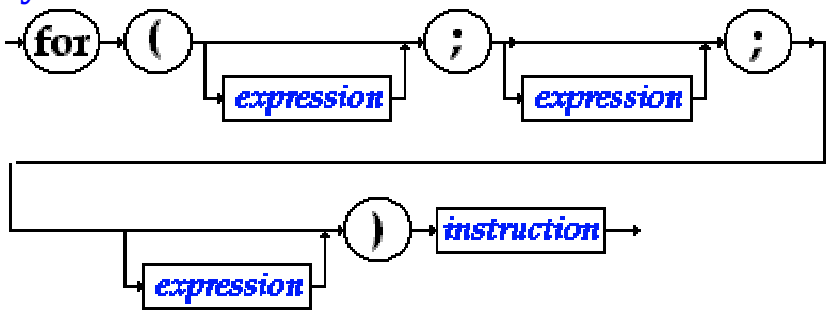
do Instr **while** (Expr) <=> Instr ; **while** (Expr) Instr

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	**for** (**i = n** ;**i !=1** ; **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),

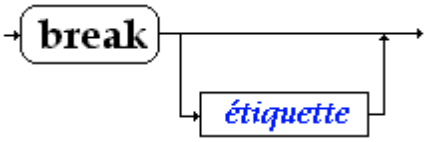
rien n'oblige à avoir une variable de contrôle,

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** :

- Si **break** n'est pas suivi d'une étiquette, elle interrompt l'exécution de la boucle dans laquelle elle se trouve, l'exécution se poursuit après le corps d'instruction.
- Si **break** est suivi d'une étiquette, elle fonctionne comme un **goto** (utilisation **déconseillée** en programmation moderne, c'est pourquoi nous n'en dirons pas plus pour les boucles !)

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

```
int [ ] table = {12,-5,7,8,-6,6,4,78,2};
int elt = 4;
for ( i = 0 ; i<8 ; i++ )
  if (elt= =table[i]) break ;
if (i = = 8)System.out.println("valeur : "+elt+" pas trouvée.");
else System.out.println("valeur : "+elt+" trouvée au rang :"+i);
```

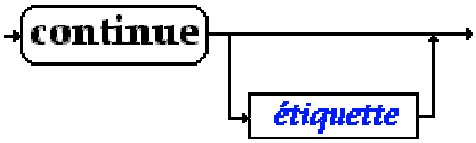
Explications

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,-6,6,4,78,2}, 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**. Lorsqu'il existe des différences avec le langage Java nous les mentionnerons explicitement**.

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** 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(String[ ] args)
  { // c'est ici que vous écrivez votre programme principal
   }
}
```

Exemple3 trivial d'une classe minimale exécutable :

```
class Exemple3
{
   static void main(String[ ] args)
  { 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,-6,6,4,78,2};
    int elt = 4, i ;
    for ( i = 0 ; i<8 ; i++ )
      if (elt= =table[i]) break ;
    if (i = = 8) System.out.println("valeur : "+elt+" pas trouvée.");
    else System.out.println("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**.

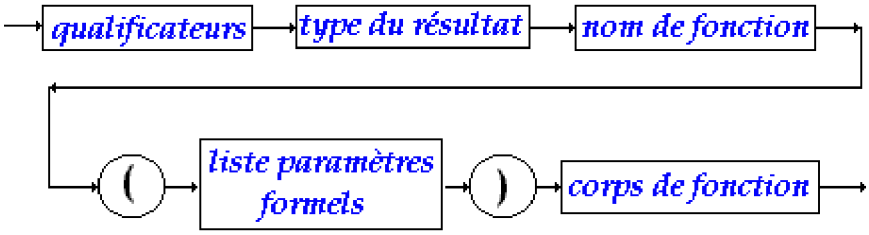
Donc par la suite dans ce document lorsque nous emploierons le mot méthode sans autre adjectif, il s'agira d'une **méthode de classe**, comme nos applications ne possèdent qu'une seule classe, nous pouvons assimiler ces méthodes aux fonctions de l'application et ainsi retrouver une *utilisation classique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

> **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 !

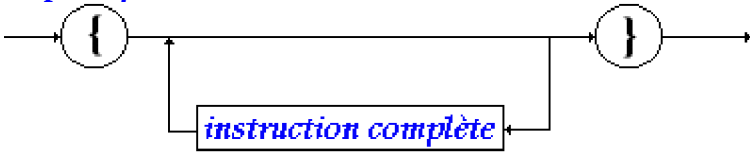
Déclaration d'une méthode

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**
boolean tester( ){.....}	renvoie un entier de type **boolean**
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**
boolean 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,-6,6,4,78,2};
      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

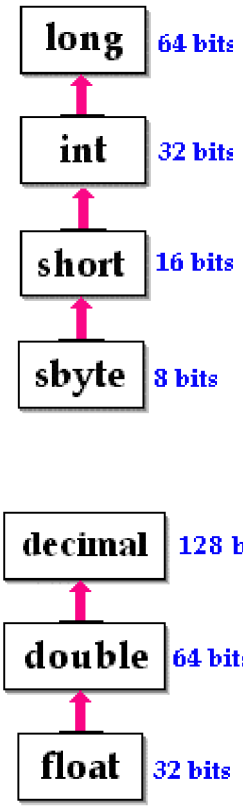
Resituons la compatibilité des types entier et réel en C#.

Un moyen mémotechnique 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**.

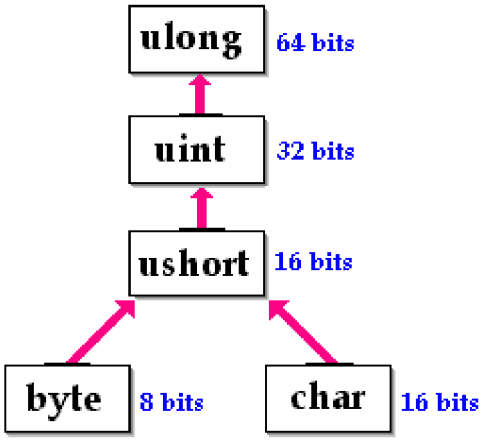
Compatibilité ascendante des types de variables en C#

(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,-6,6,4,78,2};
       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.

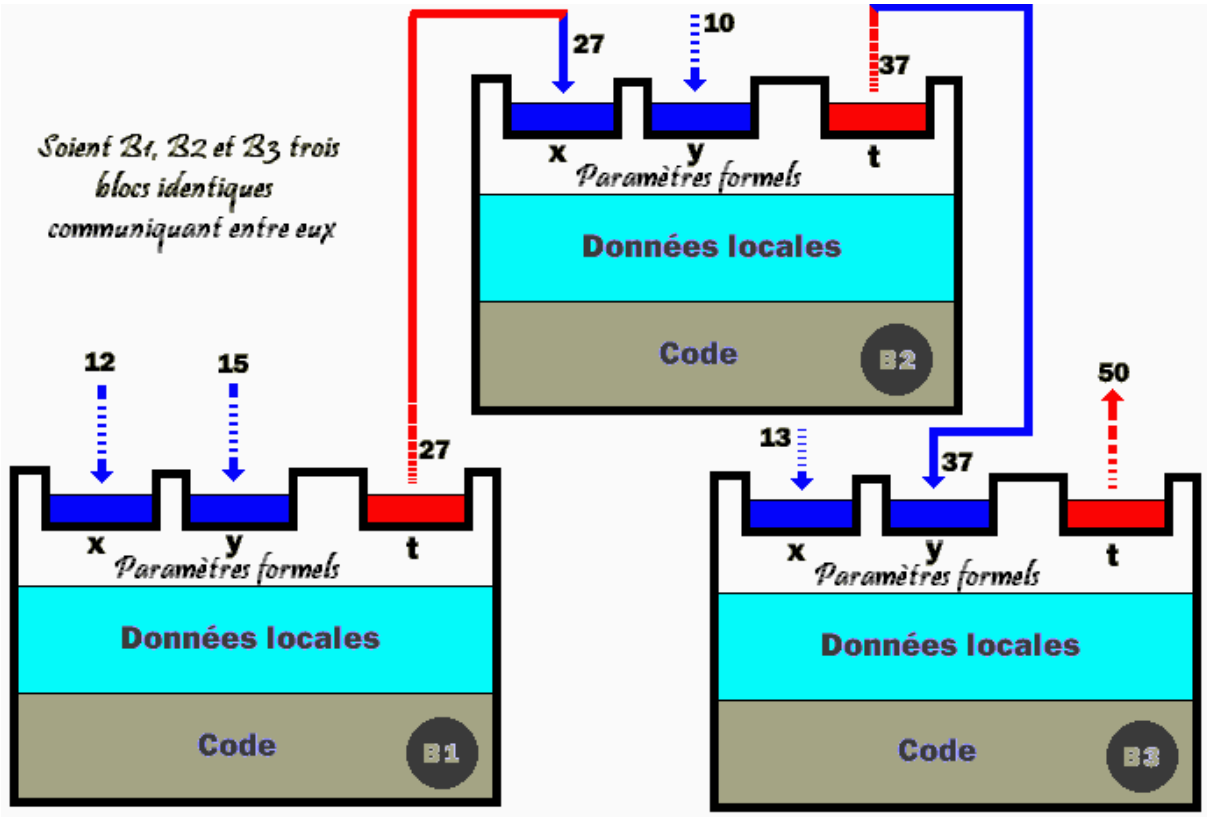
- Le paramètre effectif "**i**" est associé au paramètre formel **rang**.(**short** = entier signé sur 16 bits et **int** = entier signé sur 32 bits)

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

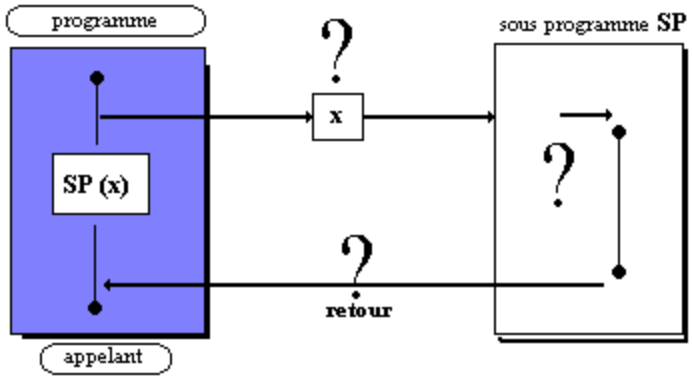
*Les **trois** modes principaux de transmission des paramètres sont :*

- 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).



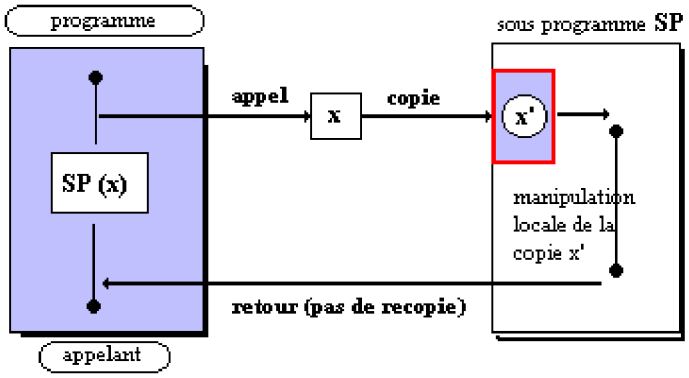
La question technique qui se pose en C# comme dans tout langage de programmation est de connaître le fonctionnement du passage des paramètres :



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**, **boolean**, **double, float**, **char**) et les objets.



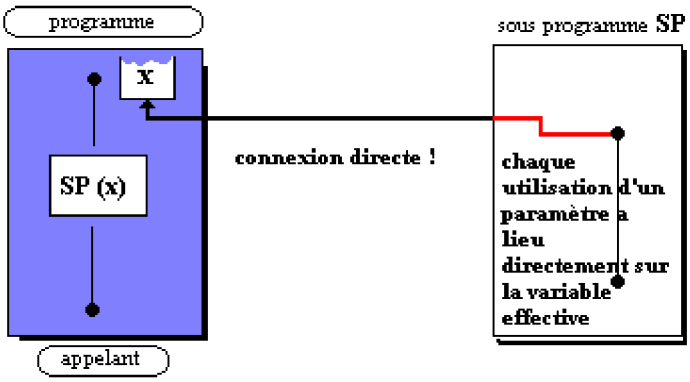
En C# tous les paramètres sont passés par défaut par valeur (lorsque le paramètre est un objet, c'est en fait **la référence de l'objet qui est passée par valeur**). Pour ce qui est de la vision algorithmique de C#, le passage par valeur permet à une variable d'être passée comme paramètre d'entrée.

```
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 :**

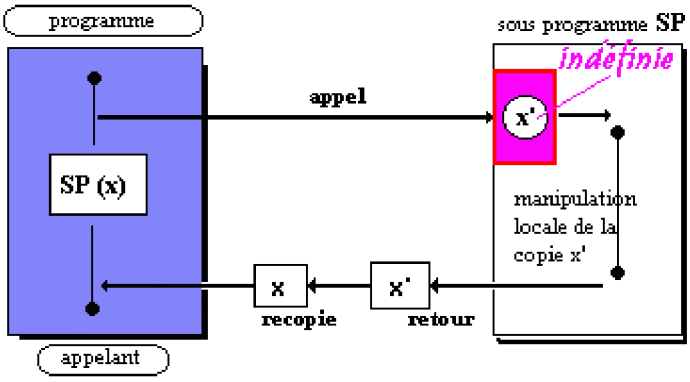
```
static int methode1(int a , ref char b) {
 //.....
  return a+b;   }
```

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**.

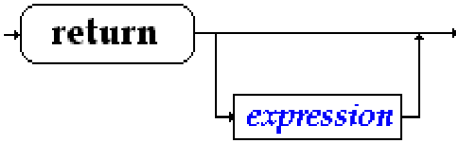
3.4 - Les retours de résultats de méthode type fonction

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

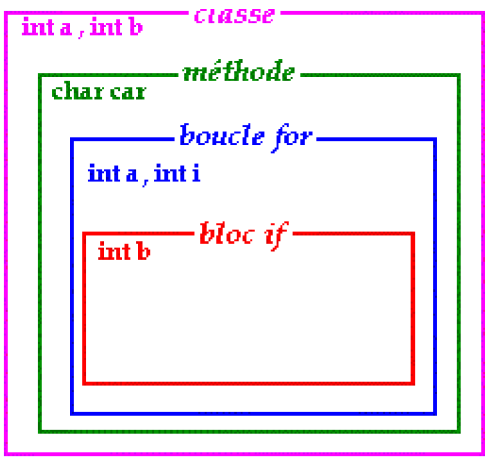
Les variables définies dans des blocs du genre instructions composées, boucles, **try…catch** ne sont visibles que dans le bloc et ses sous-blocs imbriqués, dans lequel elles sont définies.

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;
    }
}
```

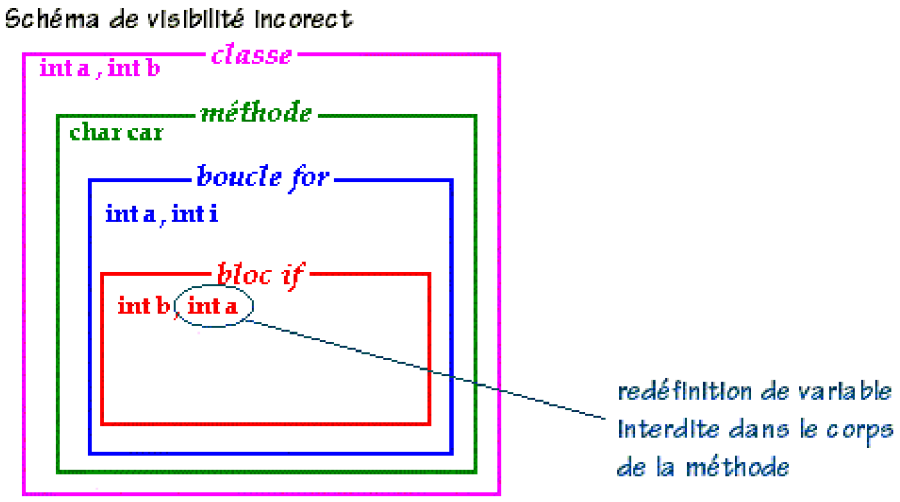
10; est masquée dans la méthode **f** dans le bloc imbriqué **for**.

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

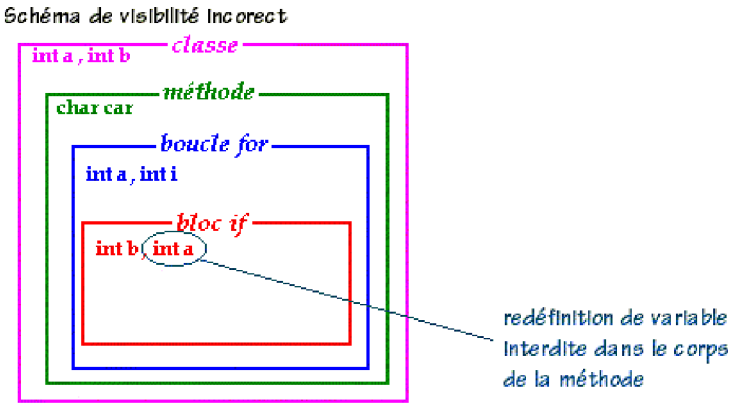
Dans l'instruction **{ int b** = 8; **b** = 5-**a**+**i***b; **}** , c'est la variable **b** interne à ce bloc qui est utilisée car elle masque la variable **b** de la classe.

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

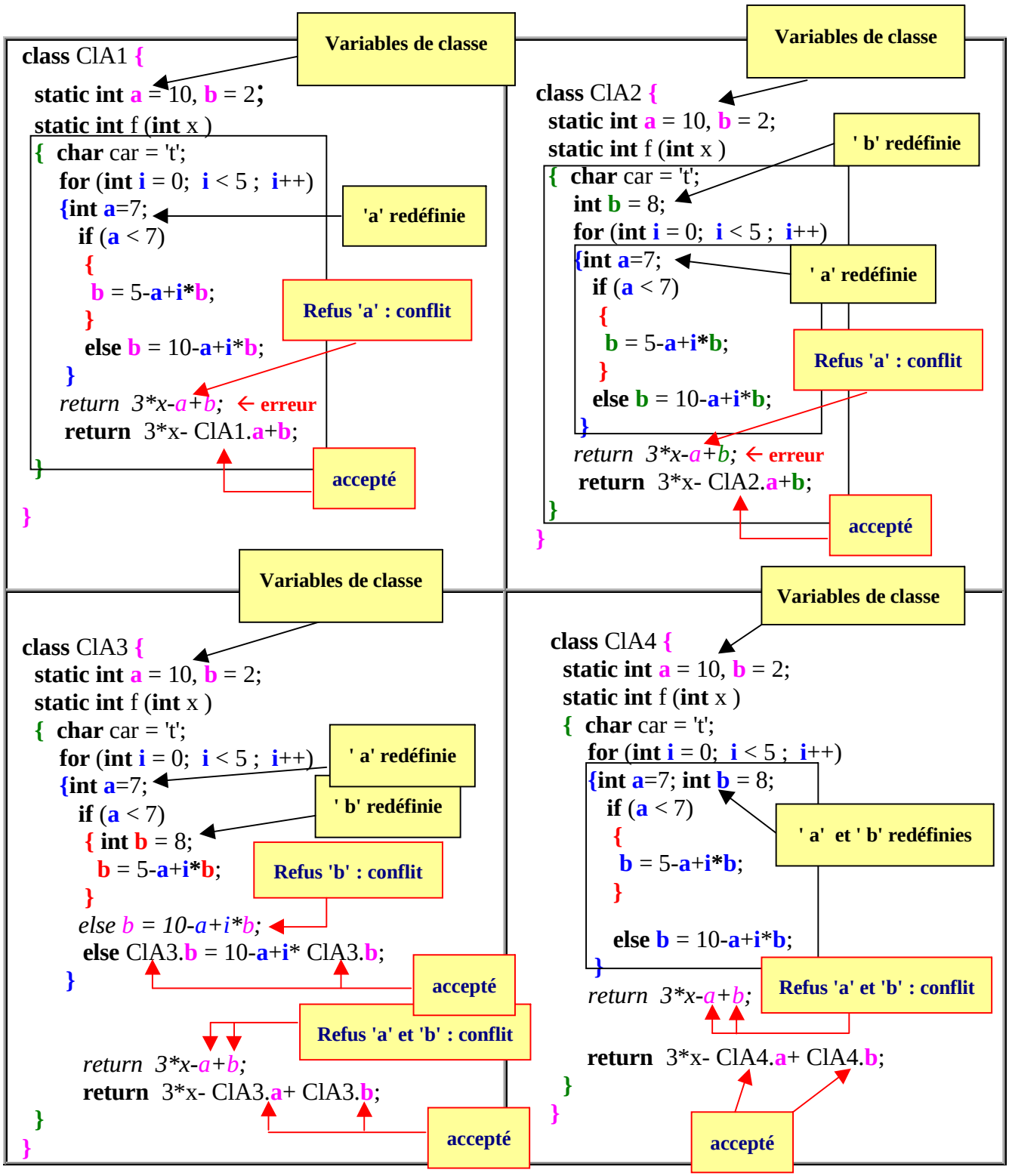
```
        return  3*x-a+b;
    }
}
```

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).

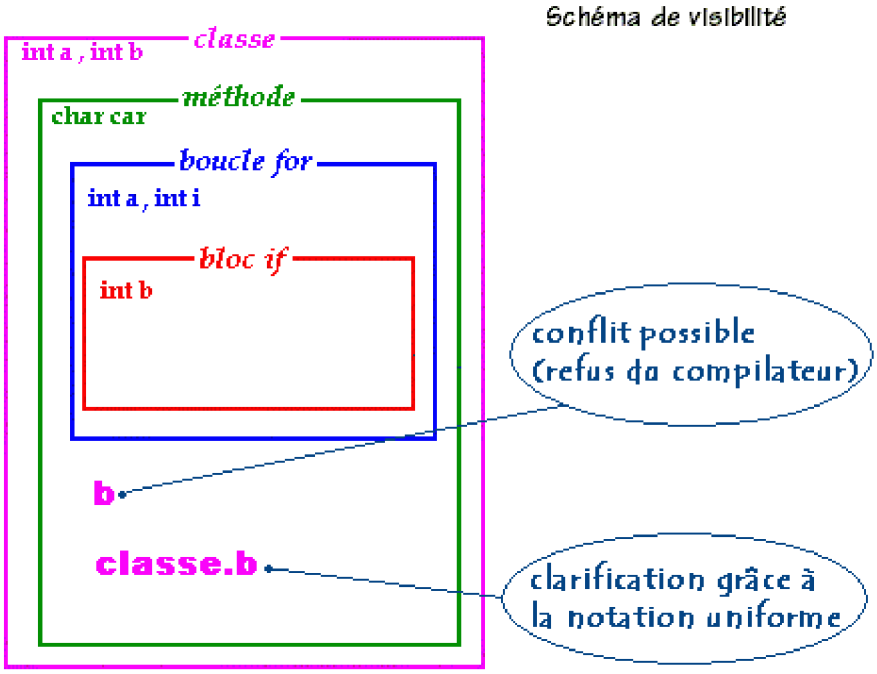
Spécificités du compilateur C#

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 :



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**.

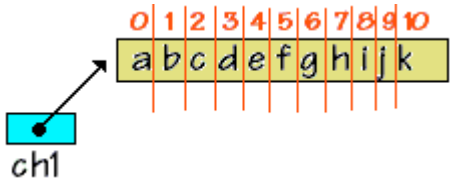
> Un littéral de chaîne est une suite de caractères entre guillemets : " abcdef " est un exemple de littéral de String.

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 "; Ou **String** str1 = **new String**("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*

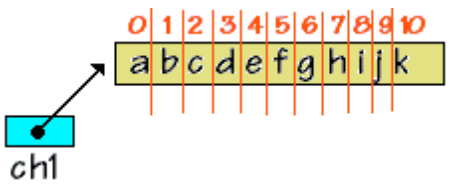
On accède à un caractère de rang fixé d'une chaîne par l'opérateur [ ] : (*la chaîne est lue comme un tableau de char*) Il est possible d'accéder en lecture seulement à chaque caractères d'une chaîne, mais qu'il est impossible de modifier un caractère directement dans une chaîne.	**String** ch1 = "abcdefghijk"; 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 caractères 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

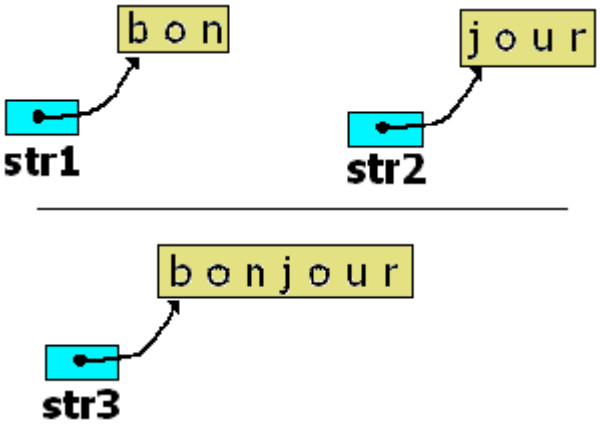
Méthode static surchargée 8 fois :

String Concat(**…**)

Les deux écritures ci-dessous sont donc équivalentes en C# :

str3 = str1+str2 ⇔ str3 = str.Concat(str2)

```
String str1,str2,str3;
str1="bon";
str2="jour";
str3=str1+str2;
```



Insertion d'une chaîne

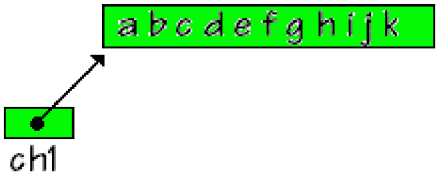
dans

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).

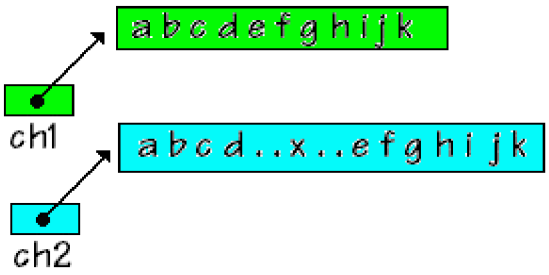
Soit :

```
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 insérereé la sous-chaîne "**..x..**".

Attention :

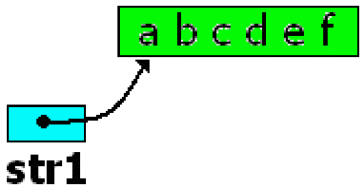
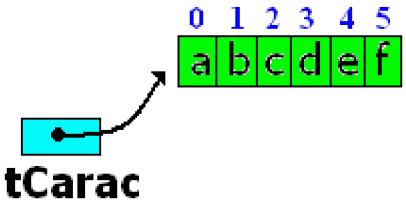
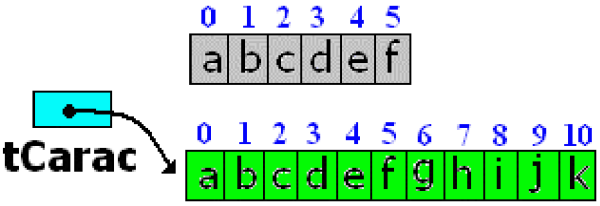
> les méthodes d'insertion, suppression, etc…ne modifient pas la chaîne objet qui invoque la méthode mais renvoient un autre objet de chaîne différent, obtenu après action de la méthode sur l'objet initial.

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,...)

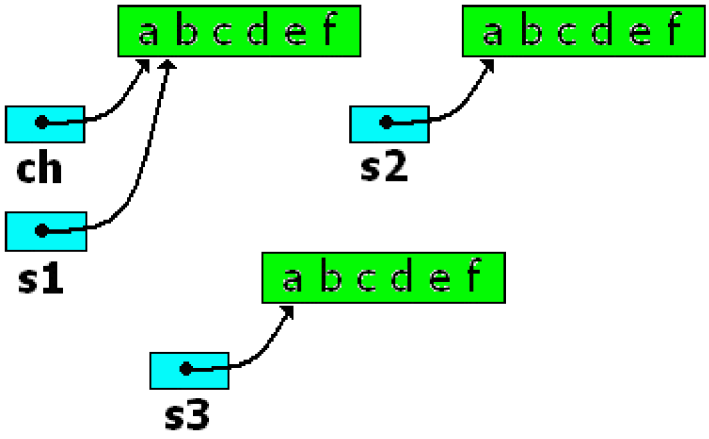
string a , b ;
(a = = b) renvoie **true** si la valeur de a est la même que la valeur de b ; sinon il renvoie **false**.

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* ——————————— s1 = 'd' + 'e'; *// types incompatibles*	**String** s1 , s2 ="abc" ; **char** c = 'e' ; s1 = "d" + **Convert**.ToString (c) ; ——————————— 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**..

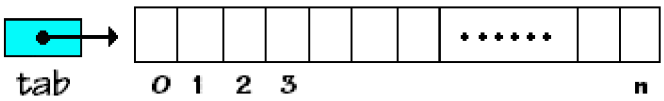
Les tableaux C# sont indexés uniquement par des entiers (**char**, **int**, **long**,…) et sur un intervalle fixe à partir de zéro. Un tableau C# peut avoir de une ou à plusieurs dimensions, nous avons donc les variétés suivantes de tableaux dans le CLR :

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 tableaux 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;
```

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

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

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 12 caractères, table3 contiendra 8 réels en simple précision et tableStr contiendra 9 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.

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 {14,-7,9}; // instancie un autre tableau de taille 3 éléments referencé par table1
```

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)

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 4)

// 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
```

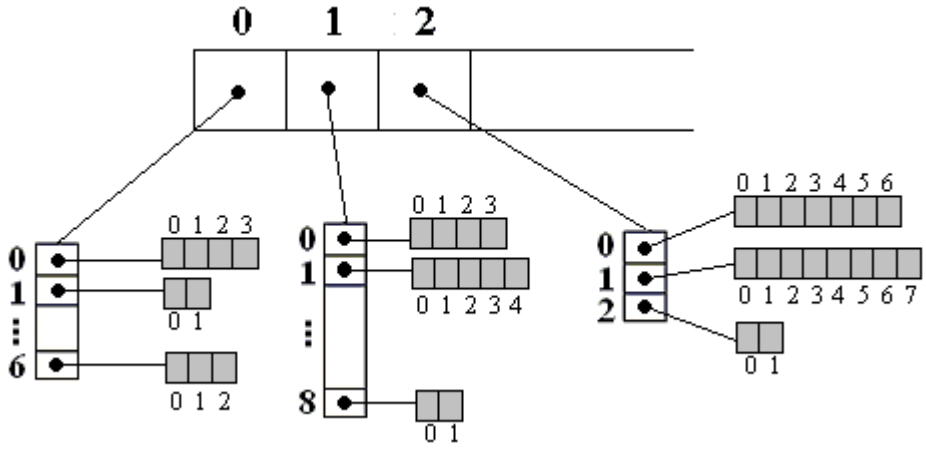
L' attribut **Length** en lecture seule, de la classe abstraite mère **Array**, contient en fait **la taille** d'un tableau en nombre total de cellules qui constituent le tableau (nous avons vu dans le cas uni-dimensionnel que cette valeur correspond à la taille de la dimension du tableau). Dans le cas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

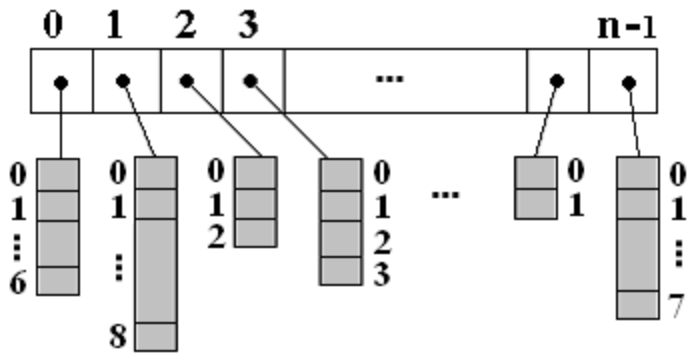
Ci-dessous le schéma d'un tableau T à trois dimensions en escalier :



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 :

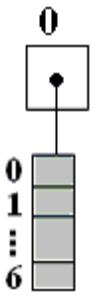
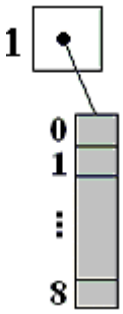
```
int n = 10;
int [ ][ ] myArray = new int [n][ ];
myArray[0] =  new int[7];
myArray[1] =  new int[9];
myArray[2] =  new int[3];
myArray[3] =  new int[4];
…
myArray[n-2] =  new int[2];
myArray[n-1] =  new int[8];
```



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 :

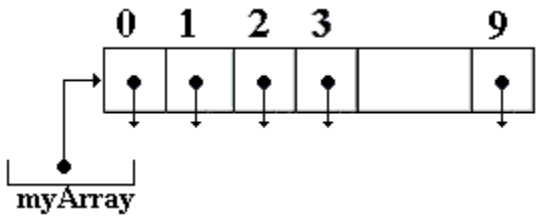
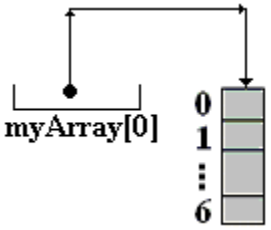
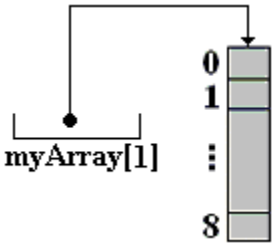
```
int n = 10;
int [ ][ ] myArray = new int [n][ ];
```

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….

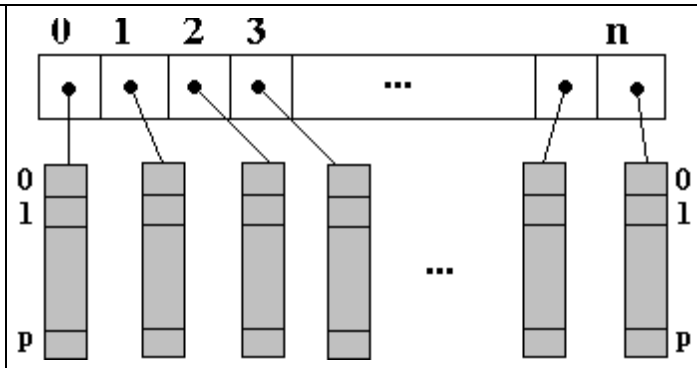
Attention

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 **[ ] [ ].**

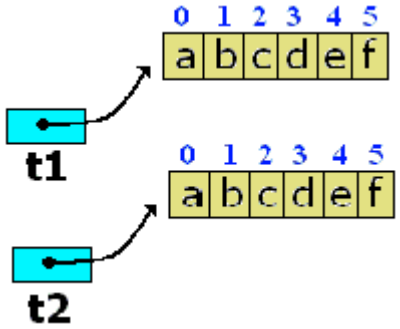
Egalité et inégalité de tableaux

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.

Affectation et recopie de tableaux

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.

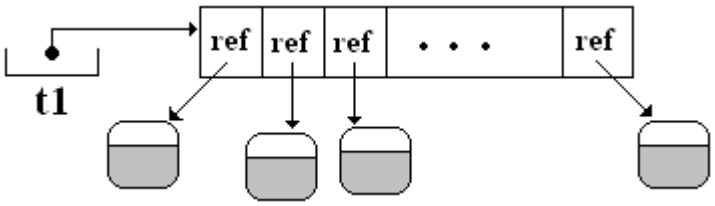
Si l'on souhaite que t1 soit une copie identique de t2, tout en conservant le tableau t2 et sa référence distincte il faut utiliser l'une des deux méthodes suivante de la classe abstraite mère Array :

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)

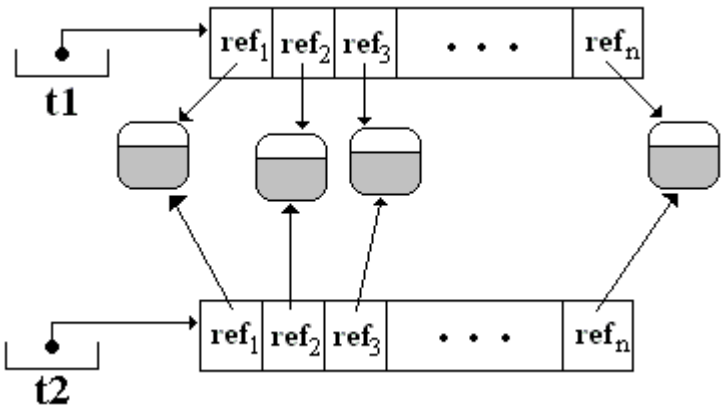
public static void **Copy** (**Array** t1 , **Array** t2, **int** long) : méthode de classe qui copie dans un tableau t2 déjà existant et déjà instancié, long éléments du tableau t1 depuis son premier élément (si l'on veut une copie complète du tableau t1 dans t2, il suffit que long représente le nombre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

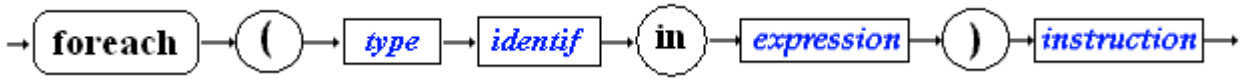
Déclaration de référence, puis clonage et transtypage.

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 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.

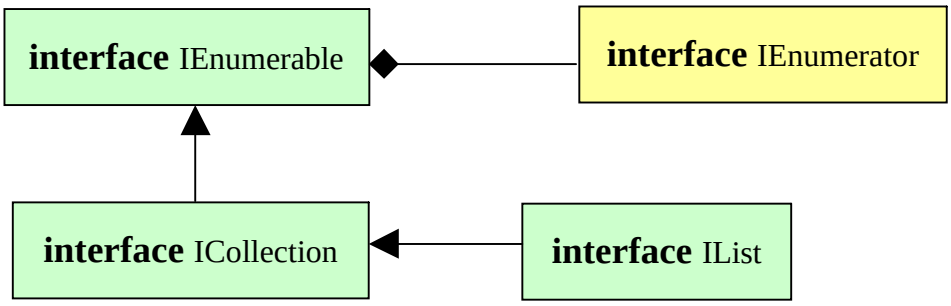
Inconvénient : on ne peut qu'énumérer en lecture les éléments d'un tableau.

Collections - Piles - Files - Listes

Généralités sur les collections

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)

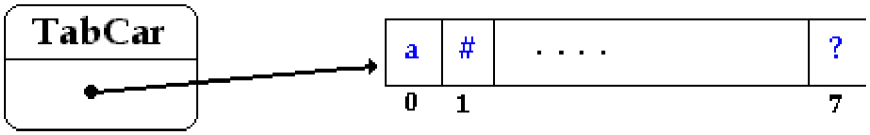
public int Add(object *elt* **);**	Ajoute l'élément *elt* à **IList**.
public void Clear( );	Supprime tous les éléments de **IList**.
public bool Contains(object *elt* **);**	Indique si **IList** contient l'élément *elt* en son sein.
public int IndexOf(object *elt* **);**	Indique le rang de l'élément *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

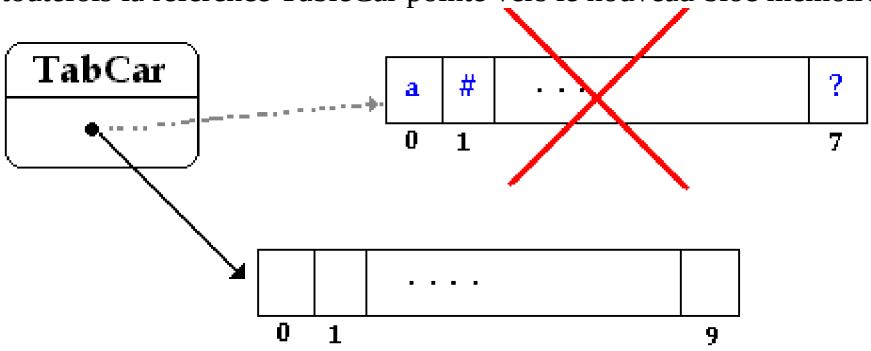
Un tabeau Array à une dimension, lorsque sa taille a été fixée soit par une définition explicite, soit par une définition implicite, **ne peut plus changer de taille**, c'est donc un objet de taille statique.

```
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;**)**

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).

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

public virtual int Add(object value);	Ajoute un 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.

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** ***:***

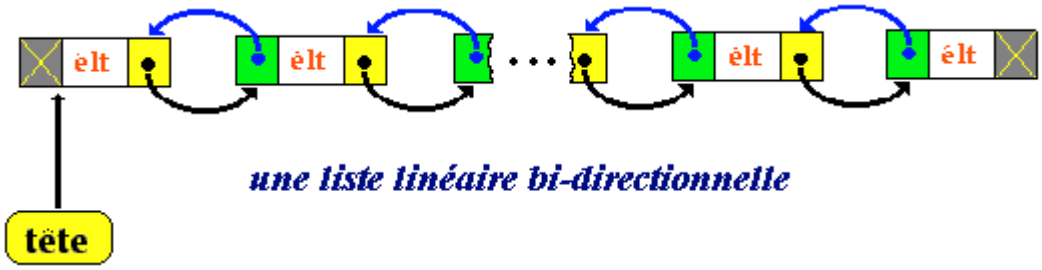
```
Vector taille = 6
Vector[0] = val:0
Vector[1] = val:1
Vector[2] = val:2
Vector[3] = val:3
Vector[4] = val:4
Vector[5] = val:5
```

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, ...). 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.

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

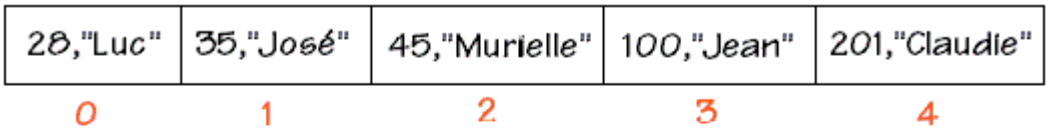
public virtual int Add(object key,object value);	Ajoute un élément avec la clé key et la valeur value spécifiées dans le **SortedList**.
public virtual void CopyTo(**Array** array, **int** arrayIndex);	Copie les éléments du **SortedList** dans une instance **Array** array unidimensionnelle à l'index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



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

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).

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.

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

System.Collections. Stack
System.Collections. Queue

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 qui ToArray 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 ;
   }
}
```

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

```
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) ;
}
```

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**, construisons de la même manière la classe Fifo de file de string dérivant de la classe Queue avec une méthode getArray et la méthode ToArray redéfinie :

```
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 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 )          {
      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) ;

      System.Console.WriteLine("----------");
      Fifo filFifo = new Fifo ( );
      filFifo.getArray(t1);
      t2 = filFifo.ToArray( );
      foreach (string s in t2)
          System.Console.WriteLine(s);
      System.Console.ReadLine();
}
```

fin
fff
eee
ddd
ccc
bbb
aaa

aaa
bbb
ccc
ddd
eee
fff
fin

Un langage très orienté objet

- Classes, objets et méthodes
- Polymorphisme d'objets
- Polymorphisme de méthodes
- Polymorphisme d'interfaces
- Classe de délégation

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 incluses dans le module, et par les classes dérivées de cette classe.
internal	Les variables et les méthodes d'une classe précédées du

	mot clef **protected** 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 **public**.

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

```
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.
```

Java

```
class Appli3Classes
{   Un x;
     Deux y;
  public static void main(String [ ]
arg)
{
    Un x;
    Deux y;
    ...
 }
 }
 class Un
 { ...
 }   class Deux
 { ...
 }
```

C#

```
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 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 méthodees 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 moins riche à ce jour qu'en Java (pas de classe membre statique, pas de classe locale et pas de classe anonyme), elle corespond 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 **public.**
- 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 **public** -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 et par défaut **public,** que le mot clef **public** 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**.

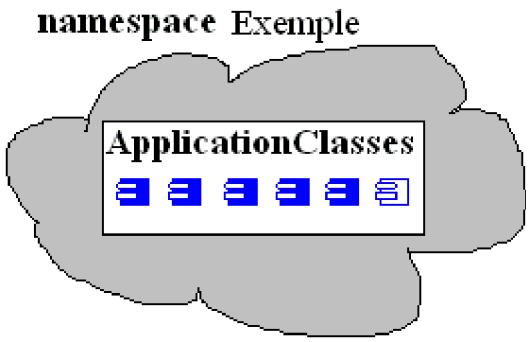
Nous étudions ci-après la visibilité des classes précédentes dans deux contextes différents.

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

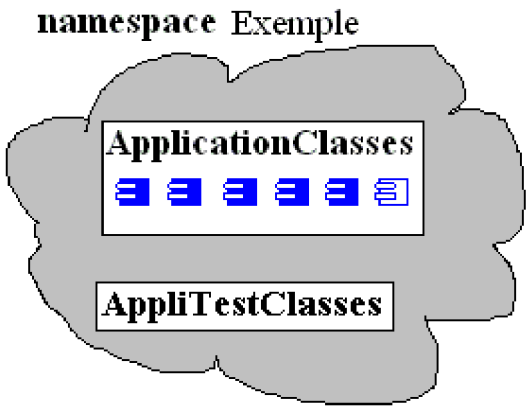



```
namespace Exemple
{
 public class ApplicationClasses {
  abstract class ApplicationClasse1 { ... }
  public class ApplicationClasse2 { ... }
  protected class ApplicationClasse3 { ... }
  internal class ApplicationClasse4 { ... }
  private class ApplicationClasse5 { ... }
  class ApplicationClasse6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

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".

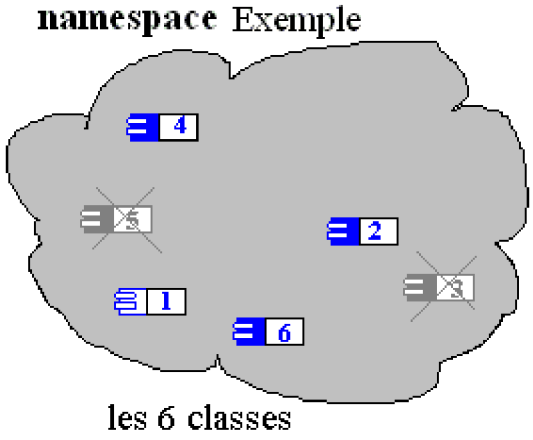
Par défaut la classe ApplicationClasse6 est ici **private**.

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

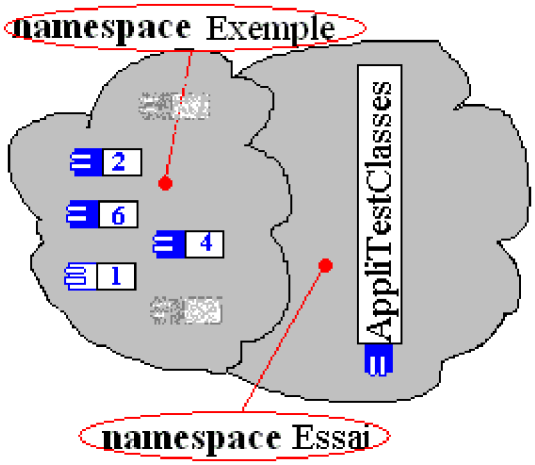
C#	Explication
	Le programme de gauche "**class** AppliTestClasses" *utilise* la classe précédente ApplicationClasses et ses sous-classes. La notation uniforme de chemin de classe est standard.
namespace Exemple { **class** AppliTestClasses { ApplicationClasses.ApplicationClasse2 a2; } }	Seule la classe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

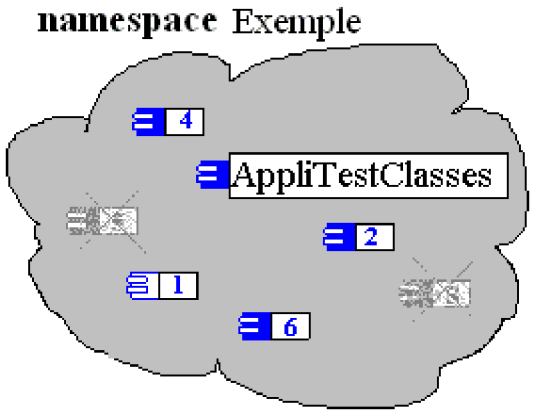
1.6 Même exemple de classes non imbriquées situées dans le même espace de noms

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 les 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 Delphi8.Net 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 **référencés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

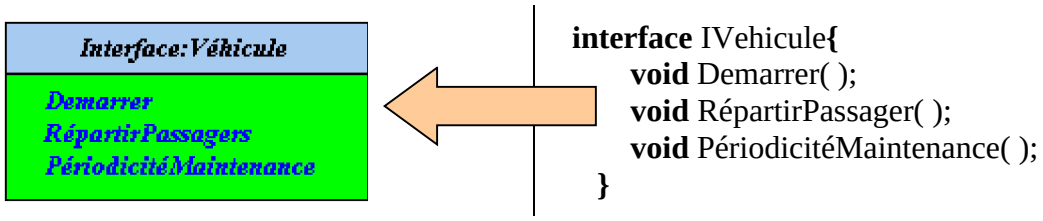
Si vous voulez utiliser la notion d'interface pour fournir un polymorphisme à une famille de classes, elles doivent toutes implémenter cette interface, comme dans l'exemple ci-dessous.

Exemple :

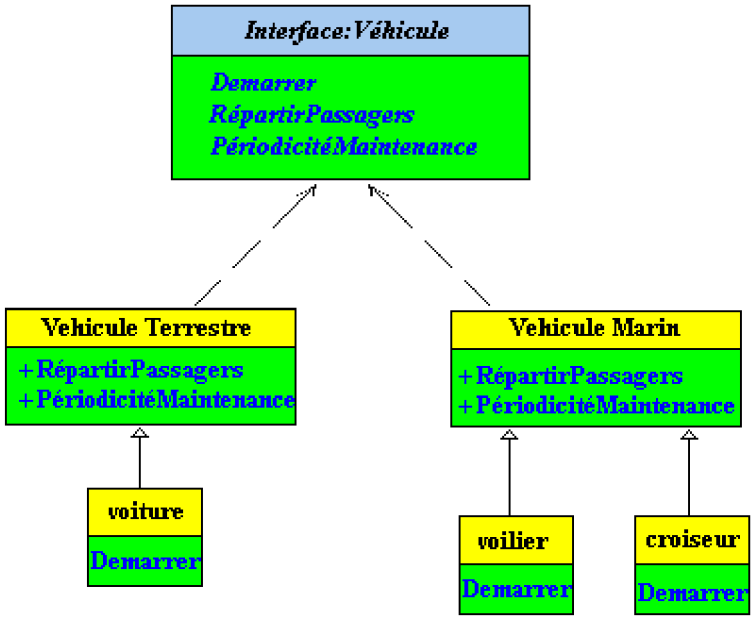
l'interface ***Véhicule*** définissant 3 méthodes (abstraites) **Démarrer**, **RépartirPassagers** de

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 partiellemnt 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 **Démarrer** de l'interface ***Véhicule*** n'est pas implémentée elles reste comme "modèle" aux futurs 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 **Démarrer** , 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 void virtual** RépartirPassager( ){...} **public void virtual** PériodicitéMaintenance( ){...} } **class** Voiture : Terrestre { **override public void** Demarrer( ){...} } **abstract class** Marin : IVehicule { **public abstract void** Demarrer( ); **public void virtual** RépartirPassager( ){...} **public void virtual** 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 void virtual** RépartirPassager( ){...} **public void virtual** PériodicitéMaintenance( ){...} } **class** Voiture : Terrestre { **override public void** Demarrer( ){...} } **abstract class** Marin : IVehicule { **public virtual void** Demarrer( ) {} **public void virtual** RépartirPassager( ){...} **public void virtual** 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.

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'effectue l'allocation et la désallocation d'objets.

Objets type valeur

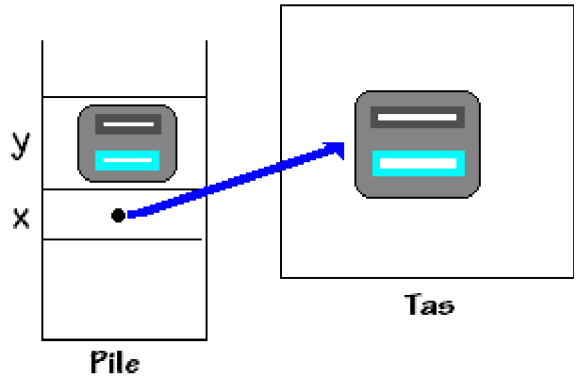
Les classes encapsulant les types élémentaires dans **.NET Framework** sont des classes de type valeur du genre **structures**. Dans le CLS une classe de type valeur est telle que les allocations d'objets de cette classe se font directement dans la pile et non dans le tas, il n'y a donc pas de référence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

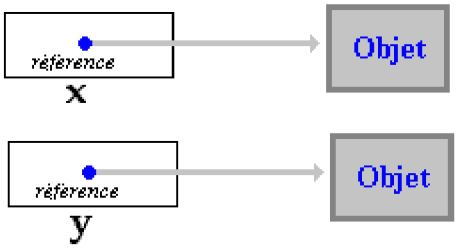
2.1 Modèle de la référence en C#

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 :

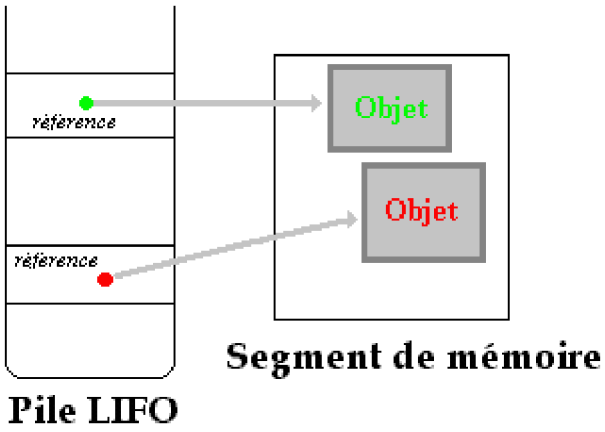
- La déclaration d'identificateur de variable typée qui contiendra la référence,
- la création de la structure de données elle-même (bloc objet de données) avec **new**.

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**( );

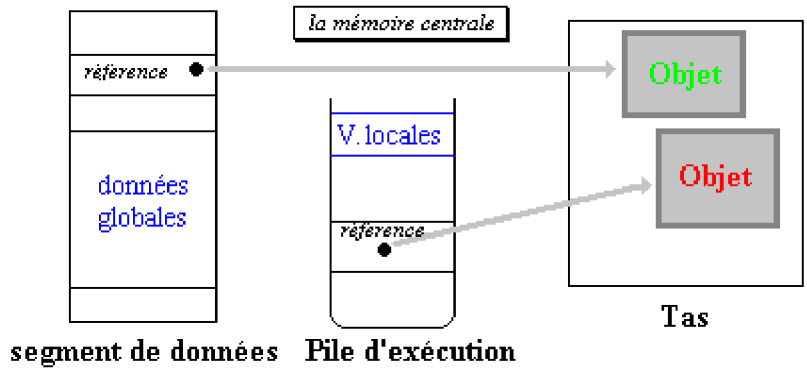
Après exécution du pseudo-programme précédent, les variables x et y contiennent chacune une référence (adresse mémoire) vers un bloc objet différent:



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 :

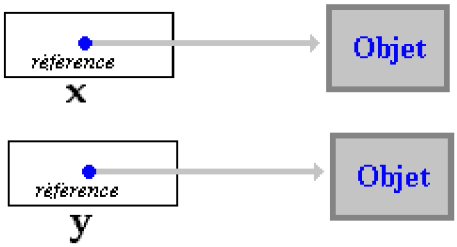


Attitude à rapprocher pour comparaison, à celle dont **Delphi** gère les objets dans une pile d'exécution de type LIFO et un tas :

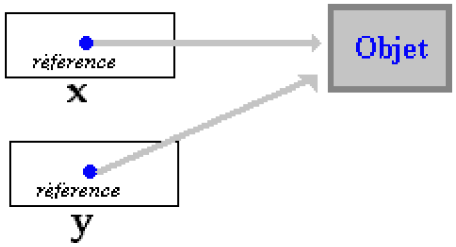


Attention à l'utilisation de l'affectation entre variables d'objets dans le modèle de représentation par référence. L'affectation x = y ne recopie pas le bloc objet de données de y dans celui de x, mais seulement la référence (l'adresse) de y dans la référence de x. Visualisons cette remarque importante :

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; } }

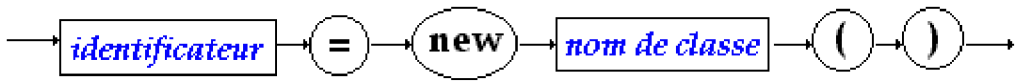
En Delphi un constructeur a un nom quelconque, tous les constructeurs peuvent avoir des noms différents ou le même nom comme en C#.

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**)

```
Un x ;
x = new Un( );
```

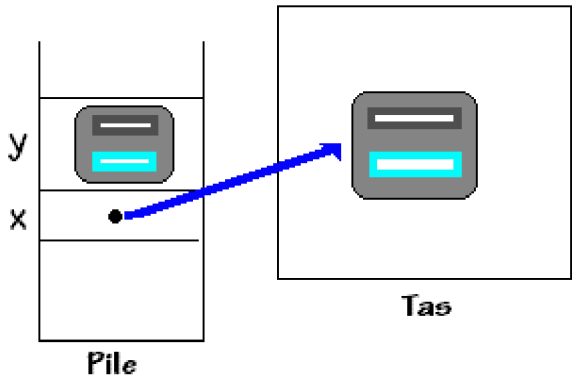
⟺

```
Un x = new 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 :

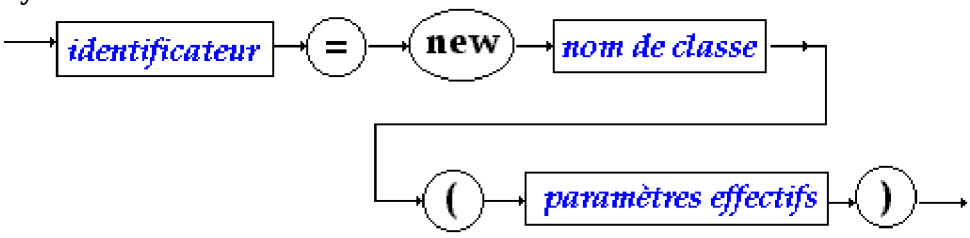
C#	Explication
class AppliClassesReferences { **public static void** Main(String [ ] arg) { Un x,y ; x = **new** Un( ); y = **new** Un( ); System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine"y.a="+y.a); y = x; x.a =12; System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a); } } **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.

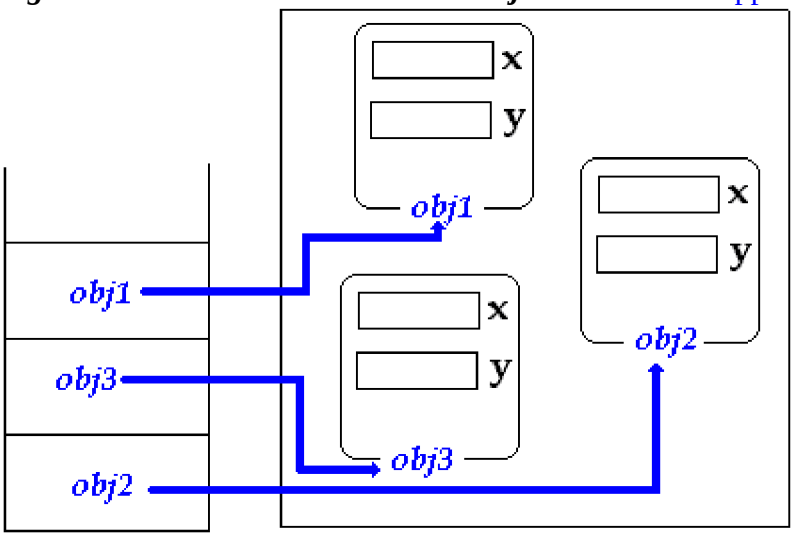
Les attributs de classe peuvent être soit de la catégorie des **variables de classe**, soit de la catégorie des **variables d'instance**.

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**.

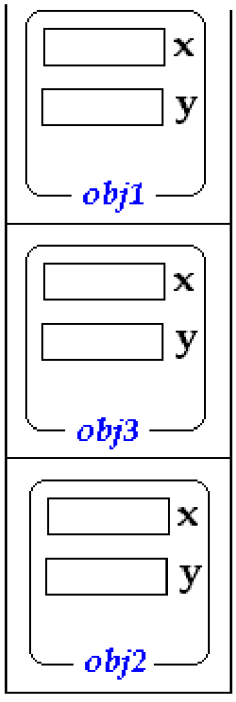
une classe C#	Instanciation de 3 objets
class AppliInstance { **int** x ; **int** y ; }	AppliInstance obj1 = **new** AppliInstance( ); AppliInstance obj2 = **new** AppliInstance( ); AppliInstance obj3 = **new** AppliInstance( );

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):

```
struct AppliInstance
{
  int x ;
  int y ;
}
```



pile

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.

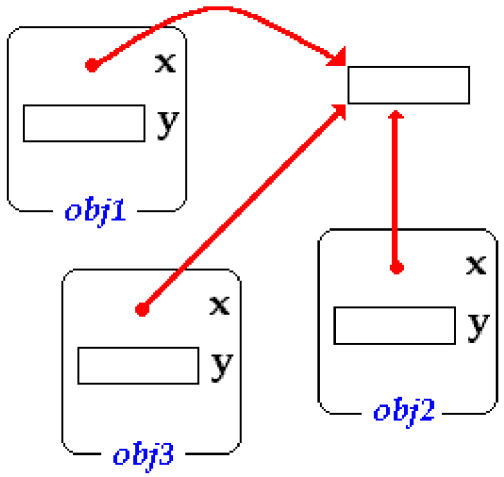
Voici deux déclaration sde variables de classe :

```
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="+obj.x); obj.g1(50); System.Console.WriteLine("<g1(50)>obj.x="+obj.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( ); AppliStatic obj2 = **new** AppliStatic( ); AppliStatic obj3 = **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 **x** de cet objet qui est modifié,les champs **x** des objets obj2 et obj3 restent inchangés

obj1.y = 100; obj1.x = 101; System.Console.WriteLine("obj1.x="+obj1.x); System.Console.WriteLine("obj1.y="+obj1.y); System.Console.WriteLine("obj2.x="+obj2.x); System.Console.WriteLine("obj2.y="+obj2.y); System.Console.WriteLine("obj3.x="+obj3.x); System.Console.WriteLine("obj3.y="+obj3.y); AppliStatic.x = 99; System.Console.WriteLine(AppliStatic.x=" +obj1.x); } }	Il y a deux manières d'accéder à la variable static **x**, : soit comme un champ de l'objet (accès semblable à celui de **y**) : obj1.**x** = 101; soit comme une variable de classe proprement dite : AppliStatic.**x** = 99; Dans les deux cas cette variable **x** est modifiée globalement et donc tous les champs **x** des 2 autres objets, 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.out.println(...) 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>obj1.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 { **static int** x = -58 ; **int** y = 20 ; **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 : **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 classe parce qu'c'est une méthode static.

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); **}** **}**	Nous avons donc **quatre** manières d'accéder à la variable static **x**, : soit comme un champ de l'objet (accès semblable à celui de **y**) : obj1.**x** = 101; soit comme une variable de classe proprement dite : AppliStatic.**x** = 99; soit par une méthode d'instance sur son champ : obj1.f1(102); soit 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.

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#

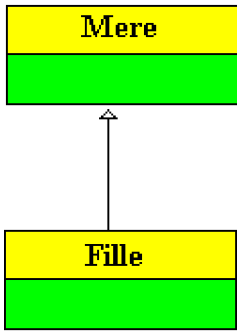
Rappel utile sur les notions de bases

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** :



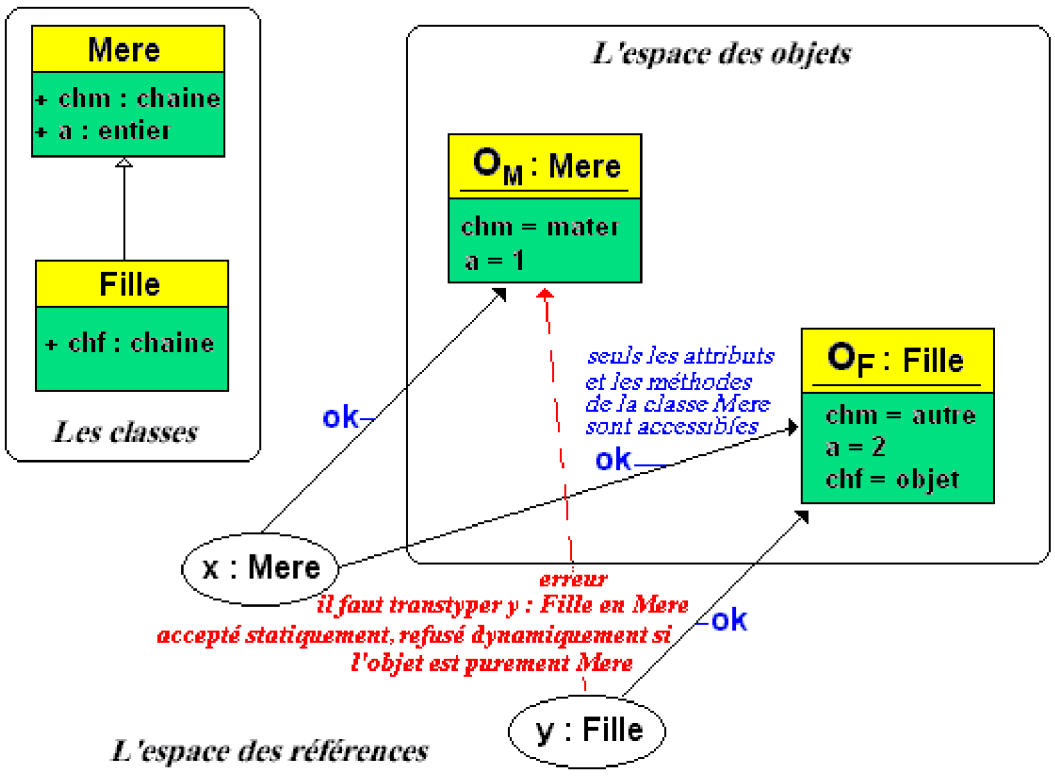
Les objets peuvent avoir des comportements polymorphes (s'adapter et se comporter différement selon leur utilisation) licites et des comportements polymorphes dangereux selon les langages.

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 quelle 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.

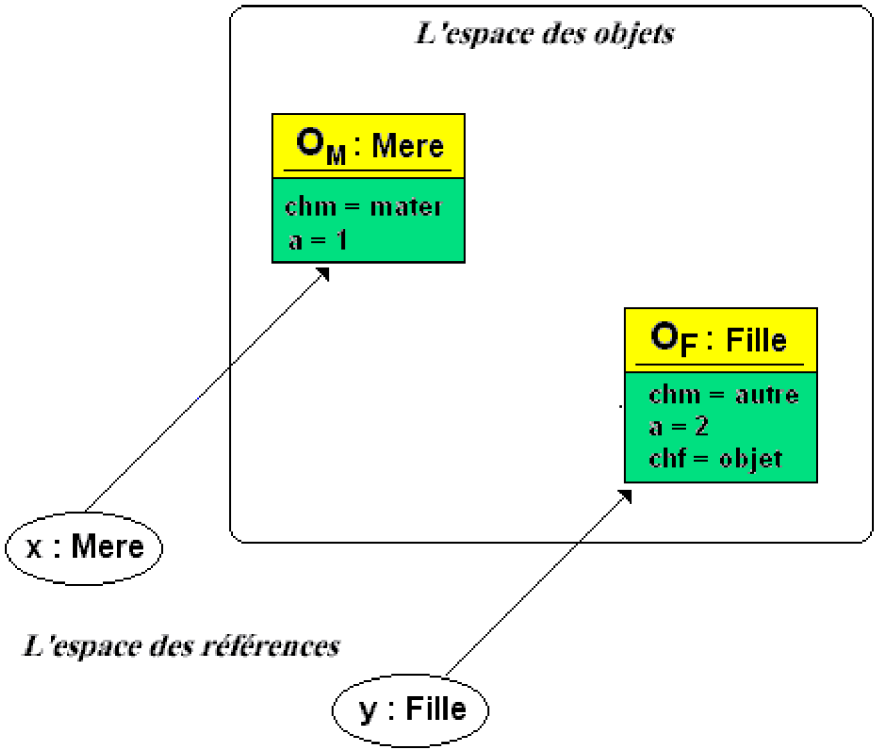


L'instanciation et l'utilisation de références dans le même type
L'affectation de références : polymorphisme implicite
L'affectation de références : polymorphisme par transtypage d'objet

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

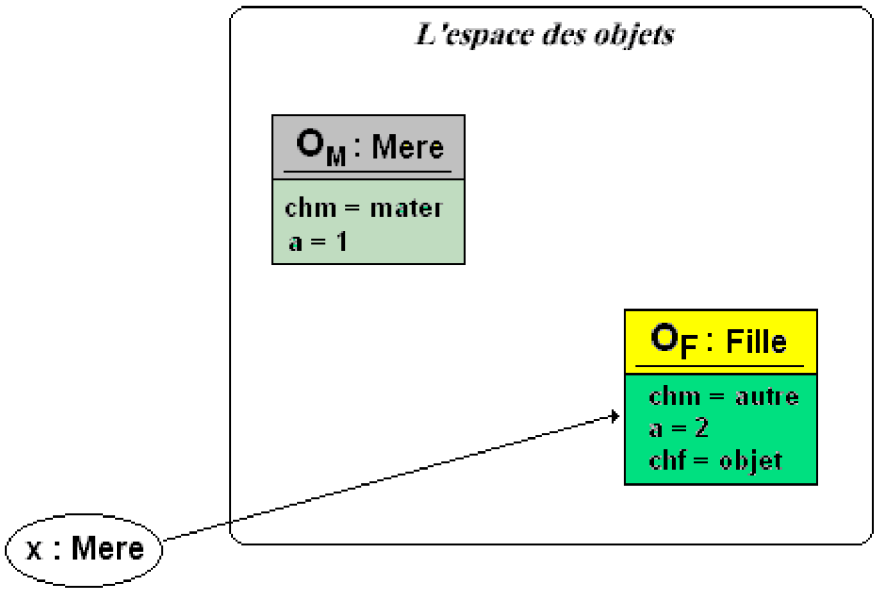
Il s'agit ici d'une utilisation la plus classique qui soit, dans laquelle une variable de référence d'objet est utilisée dans son type de définition initial (valable dans tous les LOO)



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**.

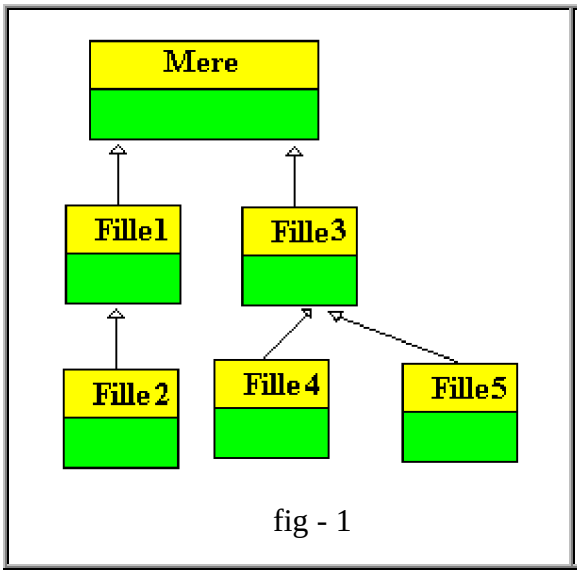


fig - 1

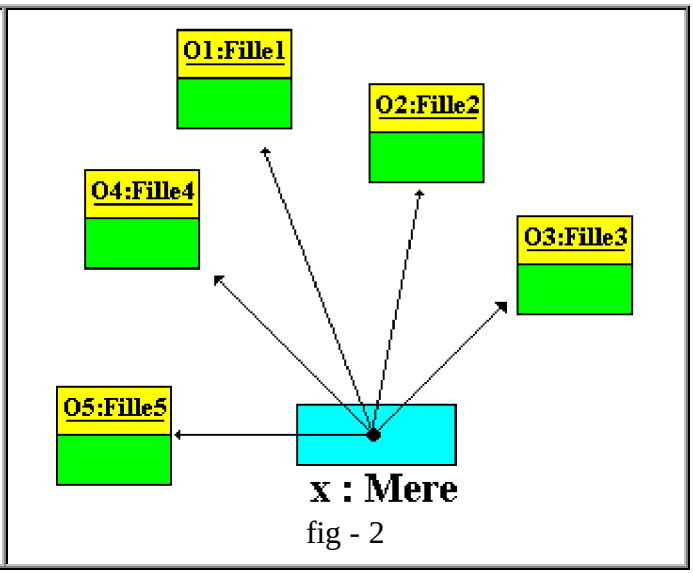


fig - 2

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 :

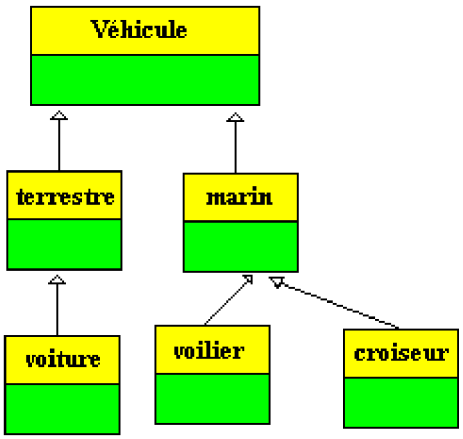
En C# :

```
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# :

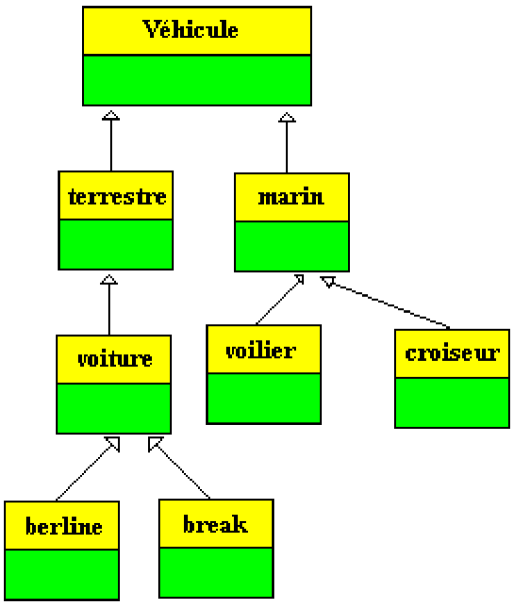
```
public class Vehicule {
  .....
}
public class terrestre : Vehicule{
 .....
}
public class voiture : terrestre {
  .....
}
```

```
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 :



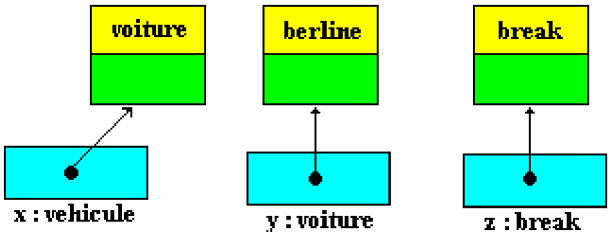
Partons de la situation pratique suivante :

- 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** Fabriquer { Vehicule x = **new** voiture ( ) ; voiture y = **new** berline ( ) ;

} **public class** break : voiture { ..... }	break z = **new** break ( ); ..... }

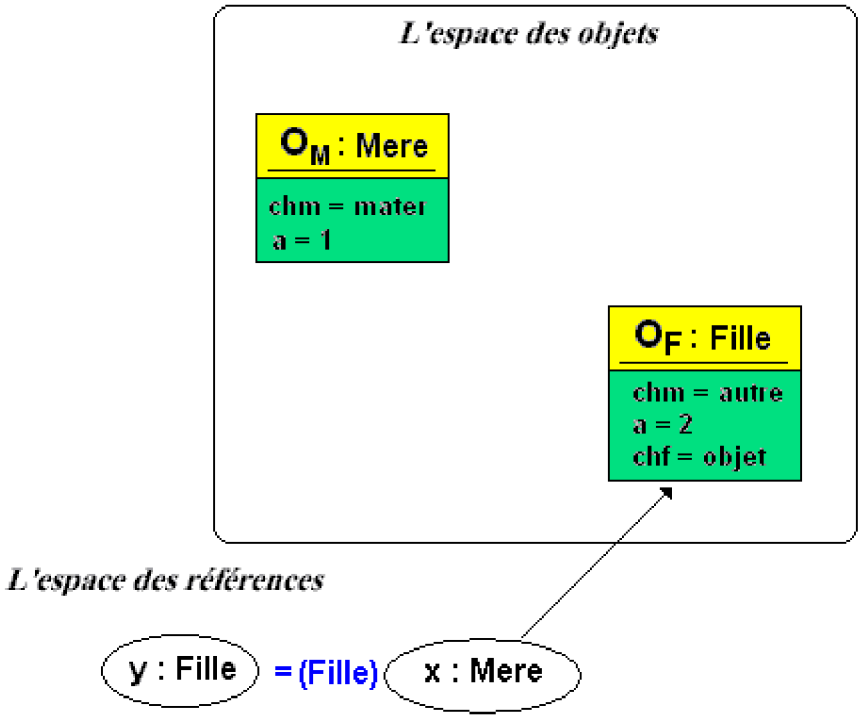
1.3 Polymorphisme d'objet explicite par transtypage

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 ;

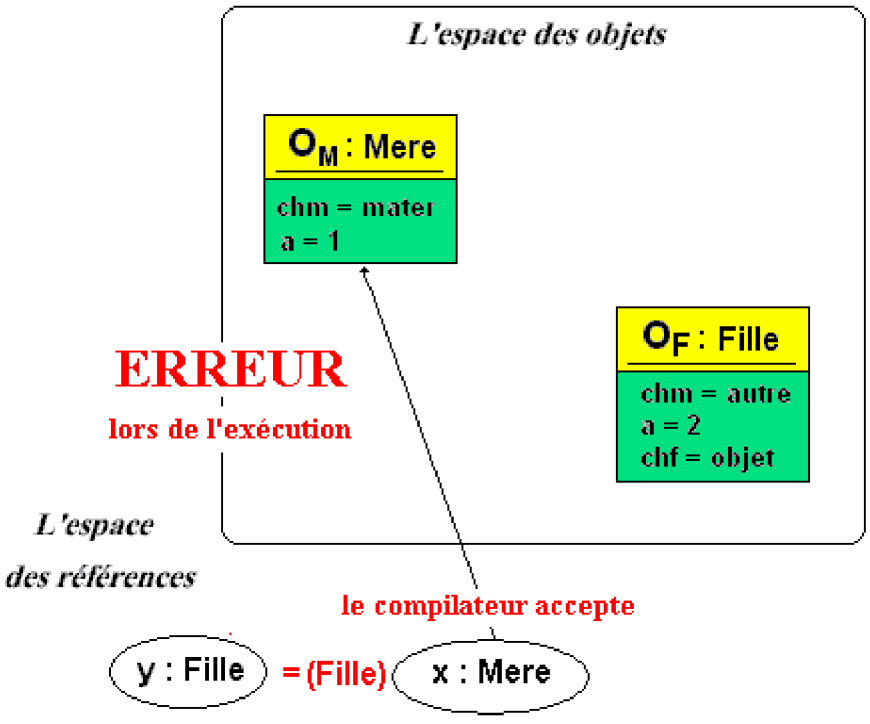
```
Fille  ObjF = new  Fille( )  ;
x = ObjF ; // x pointe vers un objet de type Fille
y = (Fille) x ; // transtypage et affectation de références du type ascendant explicite compatible dynamiquement.
```

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#).

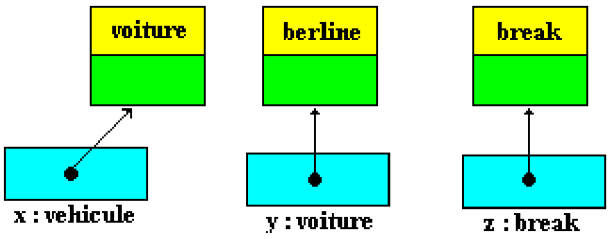
- L'opérateur **as** est un opérateur de transtypage de référence d'objet semblable à l'opérateur ( ). L'opérateur **as** fournit la valeur **null** en cas d'échec de conversion alors que l'opérateur ( ) lève une exception.




```
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 : }
```

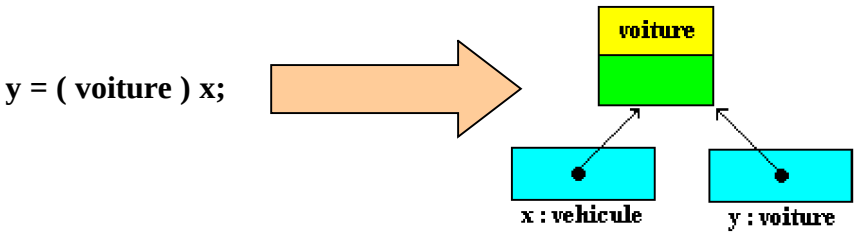
```
y = (Fille)x ;  <--- erreur lors de l'exécution ici
{affectation acceptée statiquement mais refusée dynamiquement, car x pointe vers un objet de type Mere }
```

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 :



Pour pallier à cet inconvénient de programmation pouvant lever des exceptions lors de l'exécution, C# offre au programmeur 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

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**.

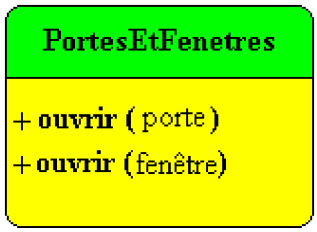
Nous avons déjà utilisé cette fonctionnalité précédement dans le paragraphe sur les constructeurs, où la classe **Un** disposait de quatre constructeurs surchargés (quatre **signatures** différentes du constructeur) :

```
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 **{** int a; **public** Un (**int** b) { a = b; } **void** f ( ) **{** a *=10; **}** **void** f (**int** x) **{** a +=10*x; **}** **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 : **void** f ( ) **{** a *=10; **}** **void** f (**int** x) **{** a +=10*x; **}** **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)

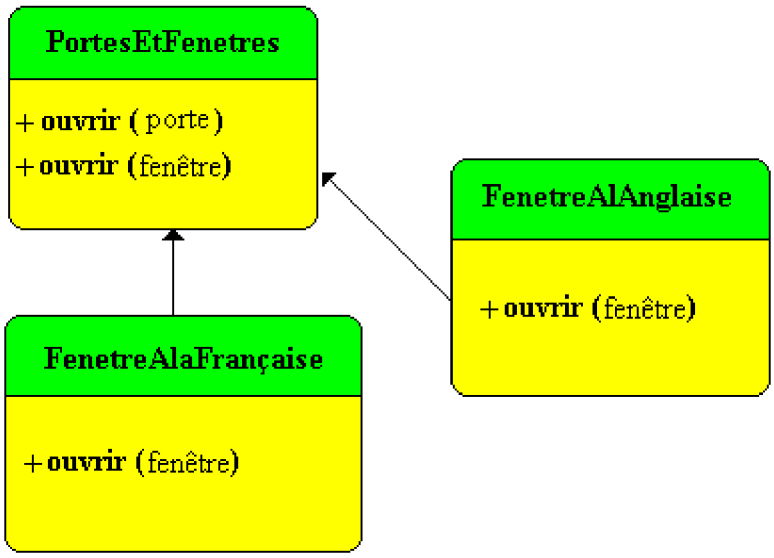
Comparaison Delphi - C# sur la surcharge :

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 **{** 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 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 laison du code, la **liaison statique** étant comme en Delphi le mode **par défaut**.

Le développeur Java sera plus décontenancé sur ce sujet, car la laison statique en Java n'existe que pour les methodes de classe **static**, de plus **la laison 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.

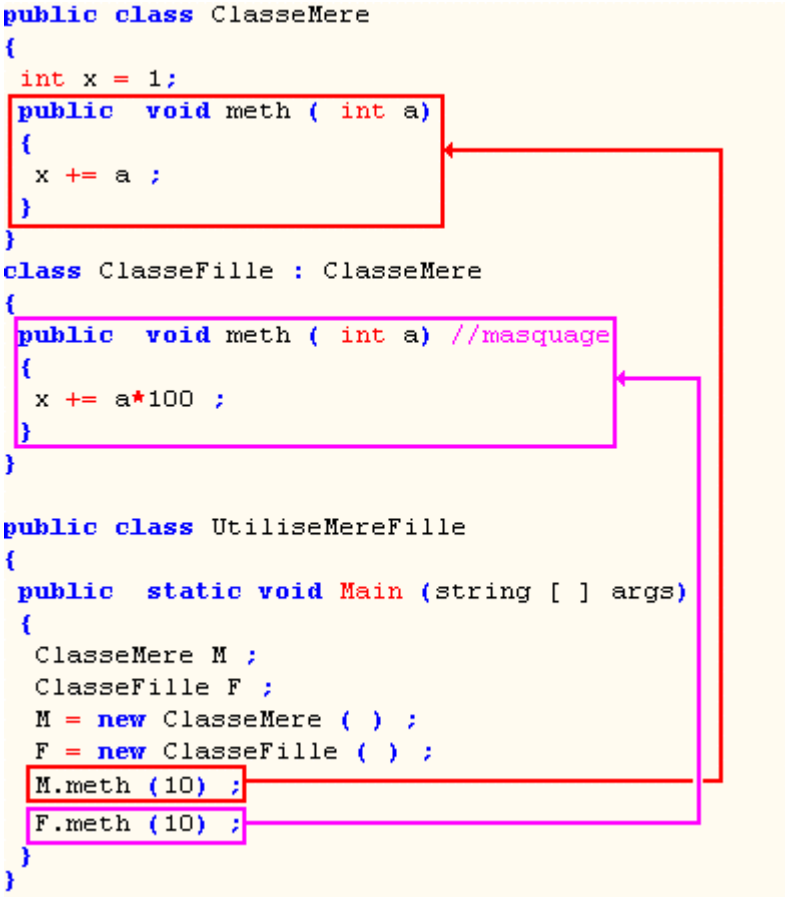
Si vous déclarez dans une classe dérivée une méthode ayant le même nom qu'une méthode statique de la classe ancêtre, la nouvelle méthode remplace simplement (on dit aussi masque) la méthode héritée dans la classe dérivée.

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 **{** **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
{
 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) ;
 }
}
```

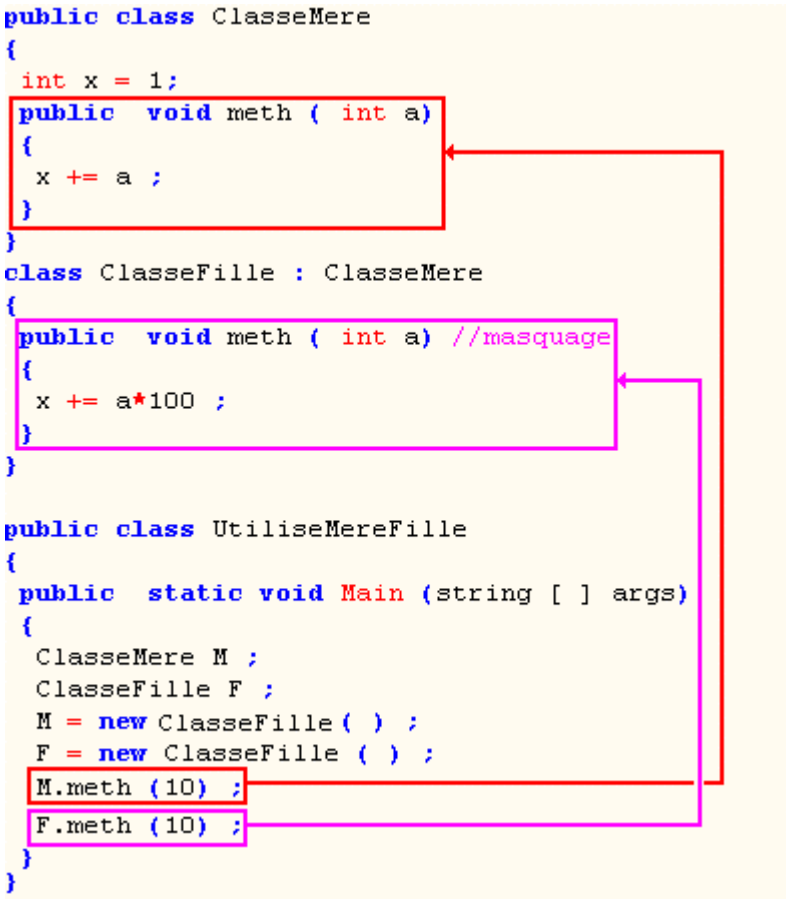
ERREUR

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...** **end**;	**class** ClasseMere **{** **int** x = 10; **public virtual void** f (**int** a) **{** x +=a; **}** **void** g (**int** a, **int** b) **{** x +=a*b; **}** **}** **class** ClasseFille **extends** ClasseMere **{** **int** y = 20; **public override void** f (**int** a) *//redéfinition* **{** x +=a; **}** **void** g1 (**int** a, **int** b) *//nouvelle méthode* **{** ..... **}** **}**

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 l'exemple précédent les instructions suivantes :
ClasseFille obj = **new** ClasseFille( );
obj.**g**(-3,8);

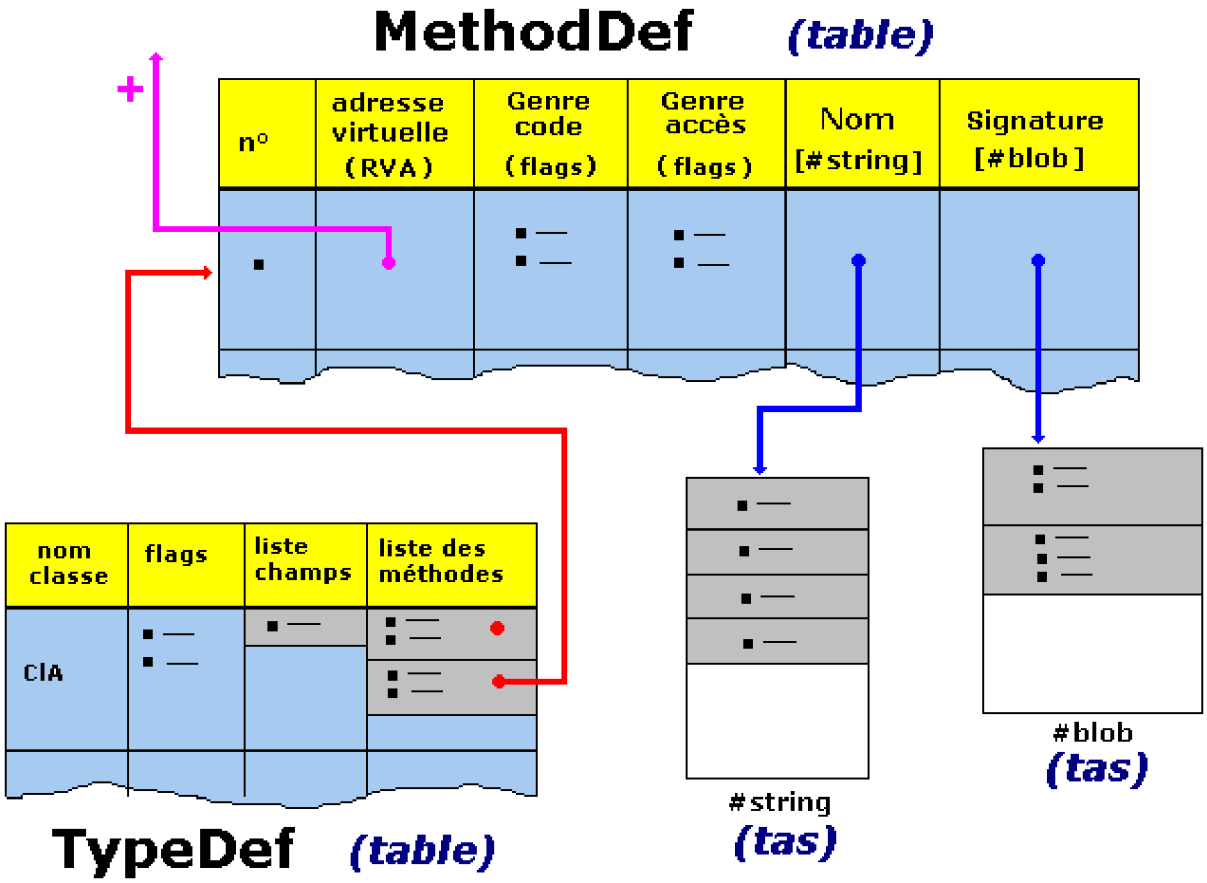
obj.**g**('h');

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 directement de faire appel à la méthode meth01 **( )** de la classe mère ClasseA car celle-ci est masquée dans la classe fille.

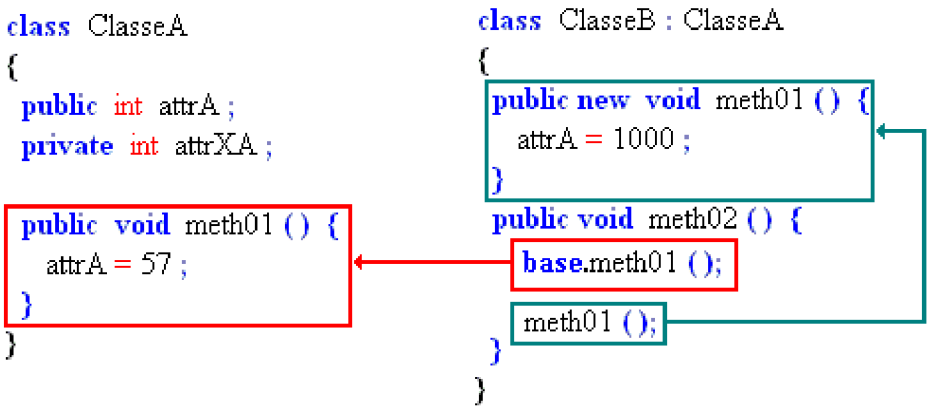
```
class ClasseB : ClasseA
{
 public new void meth01 ( ) {
   attrA = 1000 ;
 }
 public void meth02 ( ) {
   meth01 ( );
 }
}
```

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 :



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 **{**	**class** ClasseA **{**

```
  public  int  attrA ;
  public  string  attrStrA ;

}
```

```
  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 = "..." ;` `}` `/* second constructeur */` `public  ClasseB ( string  s ) {` `attrStrA = s ;` `}` `}`	`class  ClasseB : ClasseA {` `/* premier constructeur */` `public  ClasseB ( ) : base( ) {` `attrStrA = "..." ;` `}` `/* 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 ( ) **{** //..... **}** **}** 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.	**class** ClasseA **{** **public** int attrA **;** **public** string attrStrA **;** **public** ClasseA (int a) **{** **}** **}** **class** ClasseB : ClasseA **{** **public** ClasseB **( ): base( ) {** // .... **}** **}** 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 :

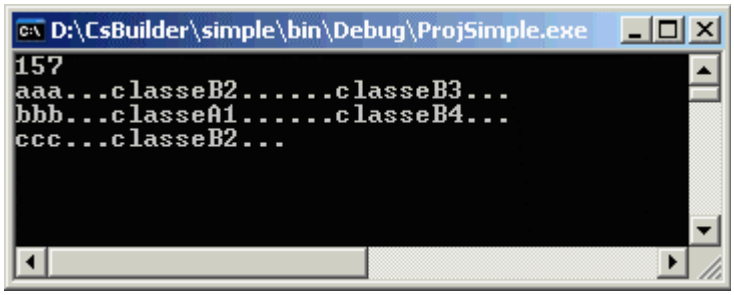
avec initialiseurs implicites-explicites	avec initialiseurs explicites équivalents
`class ClasseB : ClasseA {` `/* premier constructeur */` `public ClasseB ( ) {` `attrA = 100+attrA ;` `}` `/* second constructeur */` `public ClasseB ( string s ) {` `attrStrA = attrStrA +s+"...classeB2..." ;` `}` `/* 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..." ;`	`class ClasseB : ClasseA {` `/* premier constructeur */` `public ClasseB ( ) : base( ) {` `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) {	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

attrStrA = attrStrA+"**...classeB3...**" ; } ObjetB= **new** ClasseB(x,"aaa"); System.Console.WriteLine(ObjetB.attrStrA);	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#	Delphi

```
class ClasseA {
  public  int  attrA ;
  public  string  attrStrA ;

  public  ClasseA ( ) {
     attrA = 57 ;
  }
  public  ClasseA ( string  s ) {
     attrStrA = s +"...classeA1..." ;
  }
}
```

```
 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#	Java
`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..." ;` `}` `}`	`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 ) {` `super( ch ) ;` `attrStrA = attrStrA+"...classeB4..." ;` `}` `}`

C#	Delphi
`class ClasseB : ClasseA`	`ClasseB = class( ClasseA )`

```
{
```

```
 /* premier constructeur */
  public  ClasseB ( )
 {
     attrA = 100+attrA  ;
 }
```

```
  /* second constructeur */
  public  ClasseB ( string  s )
  {
     attrStrA = attrStrA +s+"...classeB2..." ;
   }
```

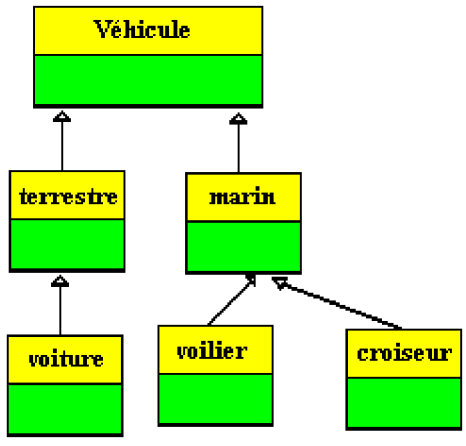
```
 /* troisième constructeur */
  public  ClasseB ( int x , string  ch ) : this( ch )
 {
     attrStrA = attrStrA+"...classeB3..."  ;
  }
```

```
 /* quatrième constructeur */
  public  ClasseB ( char x , string  ch ) : base( ch )
  {
     attrStrA = attrStrA+"...classeB4..."  ;
  }

 }
```

```
 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 :

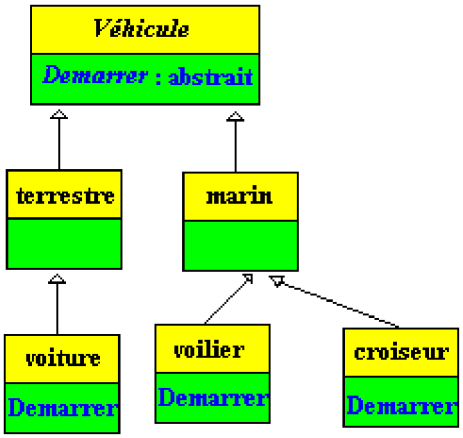
```
  class Vehicule {
```

```
class Voiture : Terrestre {
}
```

} **class** Terrestre :Vehicule **{** **}** **class** Marin :Vehicule **{** **}**	**class** Voilier : Marin **{** **}** **class** Croiseur : Marin **{** **}**

Supposons que la classe Véhicule contienne 3 méthodes, qu'elle n'implémente pas la méthode **Démarrer** 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 **Démarrer** 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 **Démarrer** 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 **Démarrer**. 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 **Démarrer** 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 **Démarrer**. 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**.

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.*

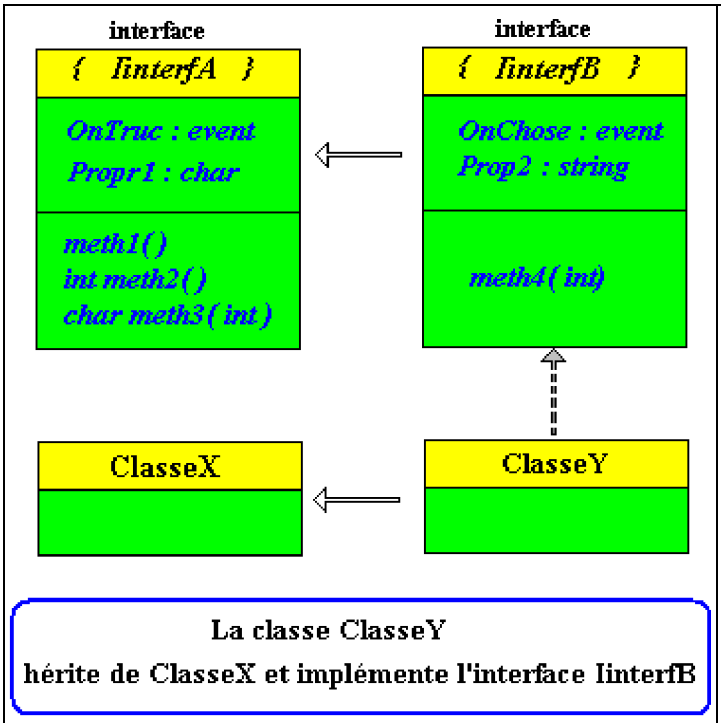
1. Vocabulaire et concepts en C#

- 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 necessite 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
}
```

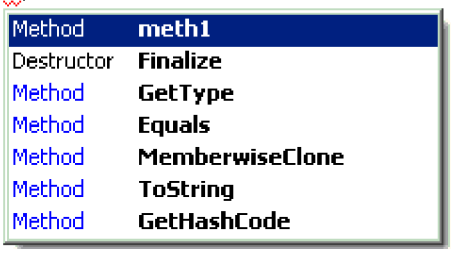
Nous implémentons l'interface IinterfA dans la classe ClasseX :

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 ( ){ ... }
}
```

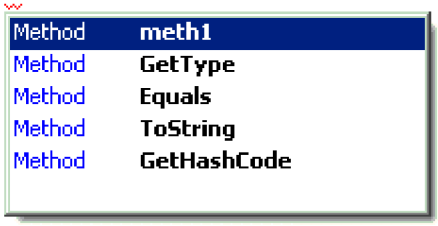
Comprenons bien que la classe ClasseX ne possède pas à cet instant une méthode d'instance qui se nommerait meth2, par exemple si dans la méthode virtuelle meth1 nous utilisons le paramètre implicite **this** qui est une référence à la future instance, l'audit de code de C#Builder nous renvoie 7 méthodes 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.}
}
```



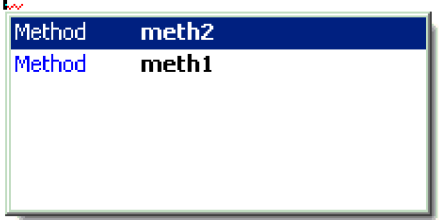
Lorsque l'on instancie effectivement un objet de classe ClasseX, cet objet ne voit comme méthode provenant de ClasseX que la méthode meth1 :

```
class Test{
 void methTest(){
      ClasseX Obj = new ClasseX();
      Obj.
 }
}
```



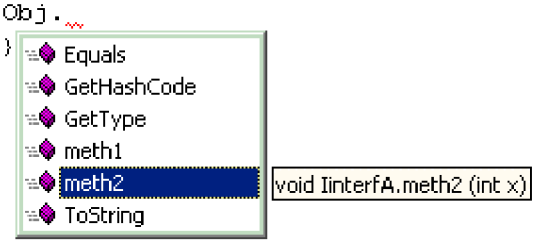
La méthode meth2 implémentée explicitement en IinterfA.meth2 devient visible uniquement si l'on utilise une référence sur l'interface IinterfA, l'exemple ci-dessous montre qu'alors les deux méthodes meth1 et meth2 sont visibles :

```
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.
 }
}
```



L'audit de code de Visual C# fournit plus de précision directement :

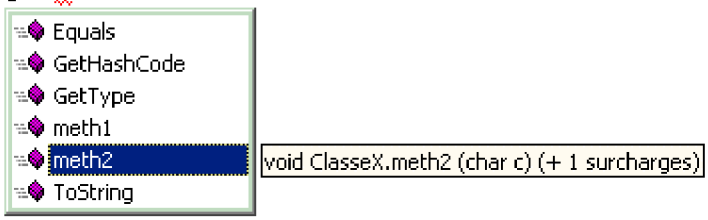
```
interface  IinterfA {
  void  meth2 ( int  x ); //  méthode de l'interface
  void  meth1 ( ); //  méthode de l'interface
}
class ClasseX :  IinterfA   {
  void  IinterfA.meth2 ( int  x ){  }
  public virtual void  meth1 ( ){ }
}
class Test  {
  void methTest()   {
    IinterfA Obj = new ClasseX();
    Obj.
    }
}
```



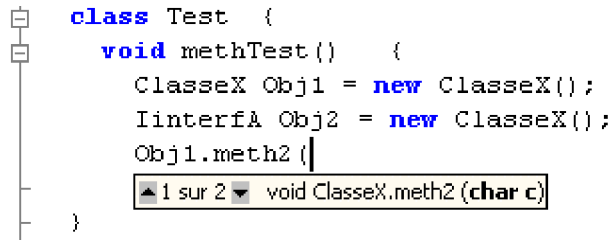
Nous voyons bien que la méthode est qualifiée avec sa signature dans IinterfA, voyons dans l'exemple ci-dessous que nous pouvons déclarer une méthode d'instance ayant la même signature que la méthode explicite, voir même de surcharger cette méthode d'instance sans que le compilateur C# n'y voit de conflit car la méthode explicite n'est pas rangé dans la table des méthodes d'instances de la classe :

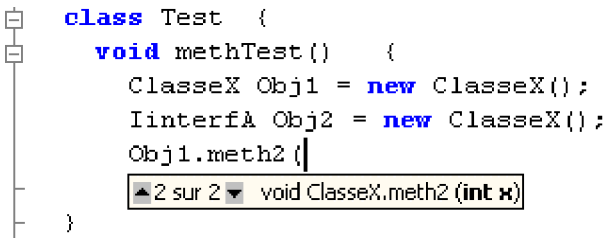
```
class ClasseX :  IinterfA {
  void  IinterfA.meth2 ( int  x ){ ... }   //méthode de IinterfA implémentée explicitement
  public virtual void  meth2 ( int  x ){ ... }  //méthode de la ClasseX surchargée
  public virtual void  meth2 ( char  c ){ ... } //méthode de la ClasseX surchargée
  public virtual 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.
    }
}
```

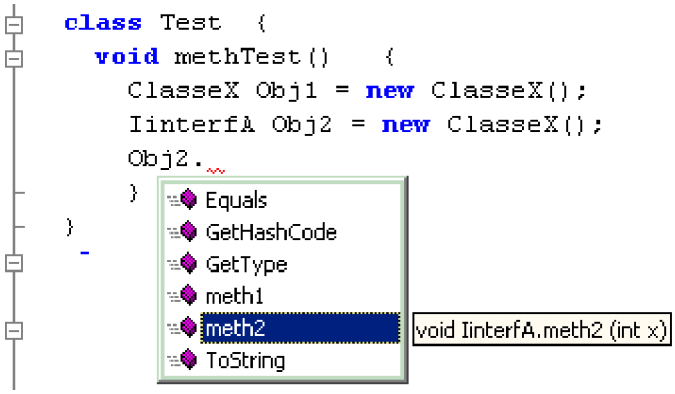


La référence Obj1 peut appeller les deux surcharges de la méthode d'instance meth2 de la classe ClasseX :





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 :



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.

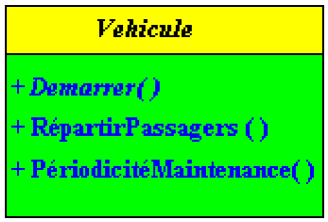
1.1 Spécification d'un exemple complet

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.**

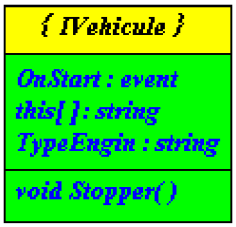
1.1.A) Une classe abstraite

La classe abstraite **Vehicule** contient trois méthodes :

classe abstraite 	• 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.

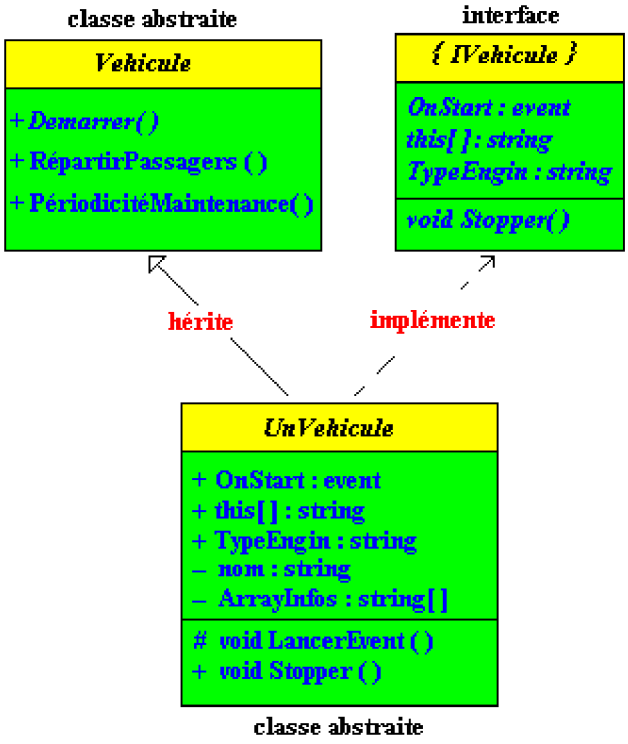
1.1.B) Une interface

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 	• 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.

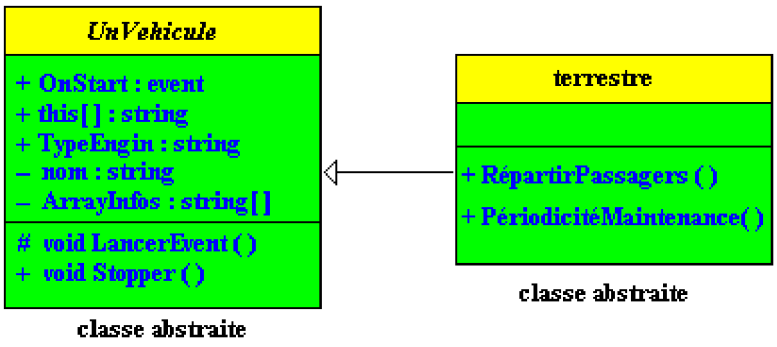
1.1.C) Une simulation d'héritage multiple

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** :



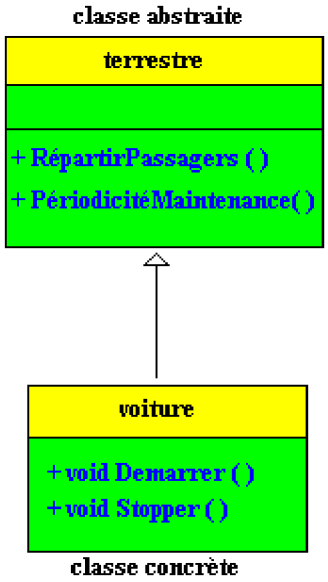
1.1.D) Encore une classe abstraite, mais plus "concrète"

Nous voulons maintenant proposer une spécialisation du véhicule en créant une classe abstraite **Terrestre** , base des futurs véhicules terrestres. Cette calsse 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** :

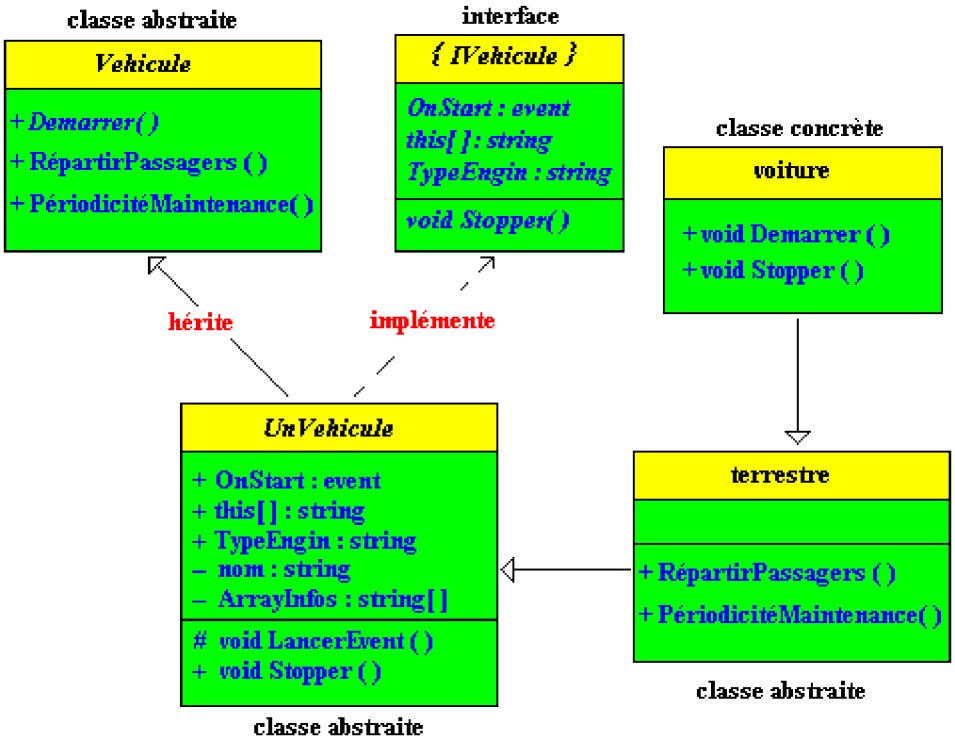


1.1.E) Une classe concrète

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**( ) :



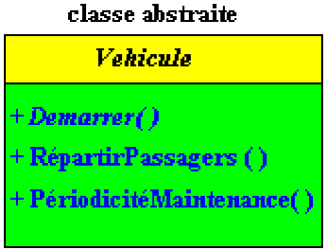
Ce qui nous donne le schéma d'héritage total suivant :



1.2 Implantation en C# de l'exemple

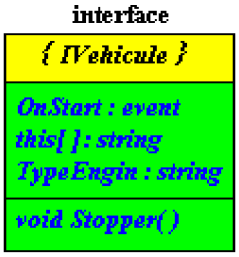
Nous proposons ci-dessous pour chaque classe ou interface une implémenation en C#.

1.2.A) La classe abstraite Vehicule



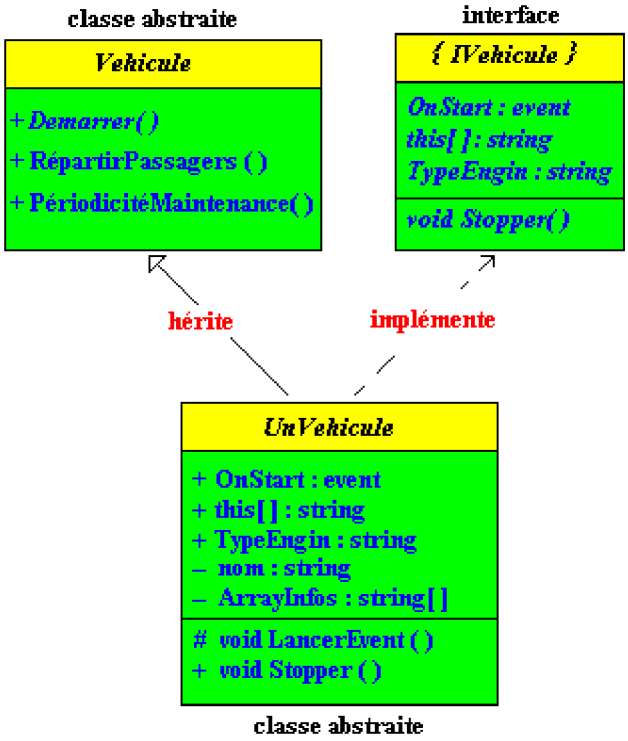

```
abstract class  Vehicule  // classe abstraite mère
{
   public abstract void  Demarrer ( ); // méthode abstraite
   public void  RépartirPassagers ( )  {  } // implantation de méthode avec corps vide
   public void  PériodicitéMaintenance ( ) {  } // implantation de méthode avec corps vide
}
```

1.2.B) L'interface IVehicule



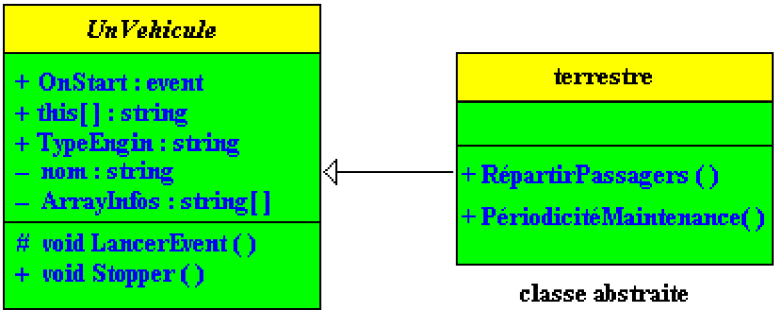

```
public delegate void  Starting ( );    //  declaration de type délégué
interface  IVehicule
{
   event  Starting OnStart ;   // déclaration d'événement du type délégué : Starting
    string  this  [ int  index]    // déclaration d'indexeur
    {
       get ;
       set ;
    }
    string  TypeEngin     // déclaration de propriété
    {
       get ;
       set ;
    }
   void  Stopper ( ); // déclaration de méthode
}
```

1.2.C) La classe UnVehicule



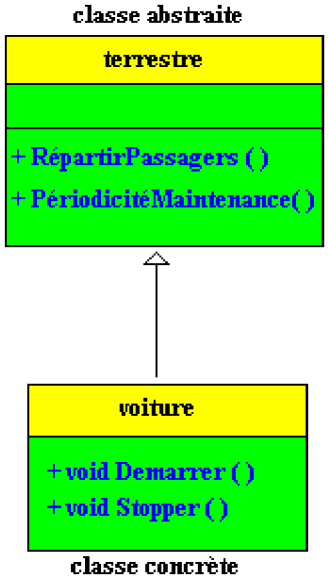

```
abstract class  UnVehicule : Vehicule , IVehicule  // hérite de la classe mère et implémente l'interface
{
  private  string  nom = "";
  private  string [ ] ArrayInfos = new  string [10] ;
  public  event  Starting OnStart ;
  protected void  LancerEvent ()
  {
    if( OnStart  !=  null)
       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.

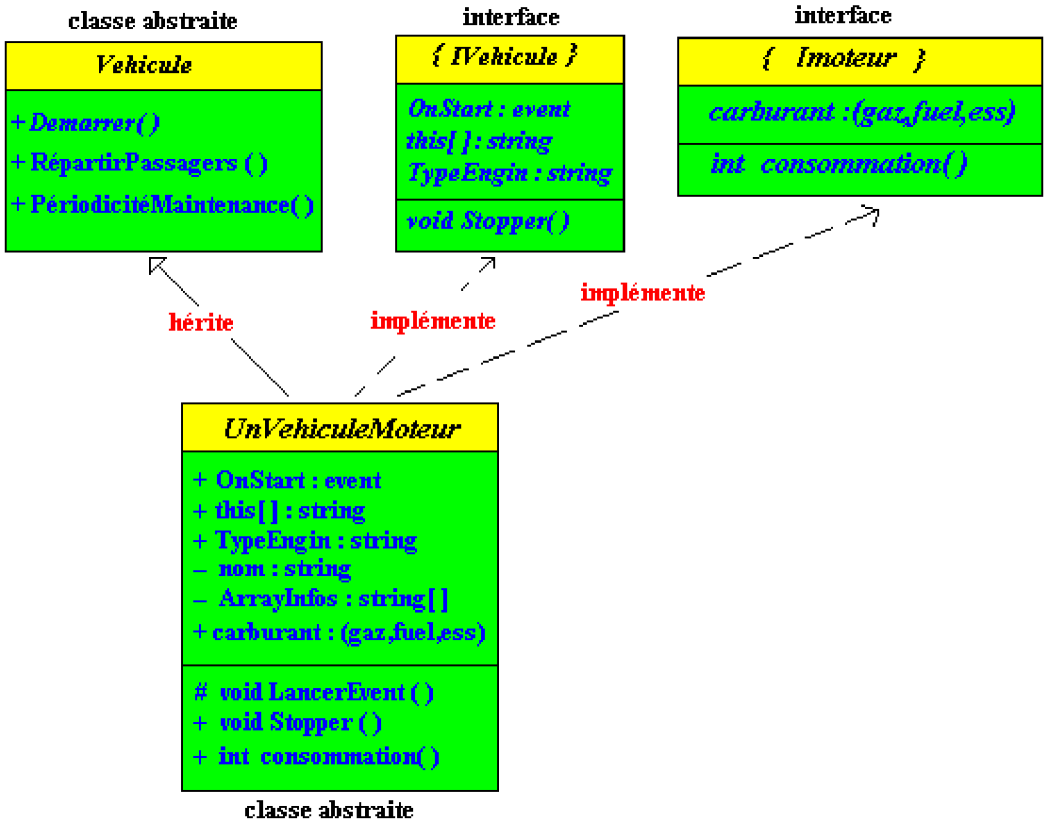
Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui ne devraient plus être modifiés. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

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 **référencés** 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 ;
  }
}
```

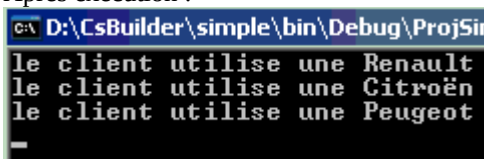
```
  c1 = new Voiture( );
  c1.TypeEngin = "Peugeot";
  UseVoiture1 client = new UseVoiture1( );
  ch = s+client.use(a1);
  System.Console.WriteLine(ch);
  ch = s+client.use(b1);
  System.Console.WriteLine(ch);
  ch = s+client.use(c1);
  System.Console.WriteLine(ch);
 }
}
```

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 :



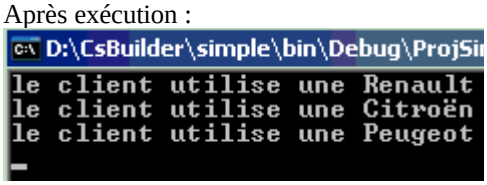
a1 est une référence sur IVehicule, b1 est une référence sur une interface fille de IVehicule, c1 est de classe Voiture implémentant IVehicule.

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 :



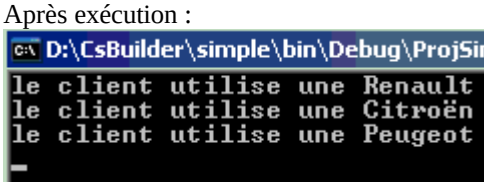
Le polymorphisme de référence d'objet appliqué à la méthode **public string** use (**UnVehiculeMoteur** x) indique que les paramètres passés doivent être de type UnVehiculeMoteur ou de type descendant.

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).

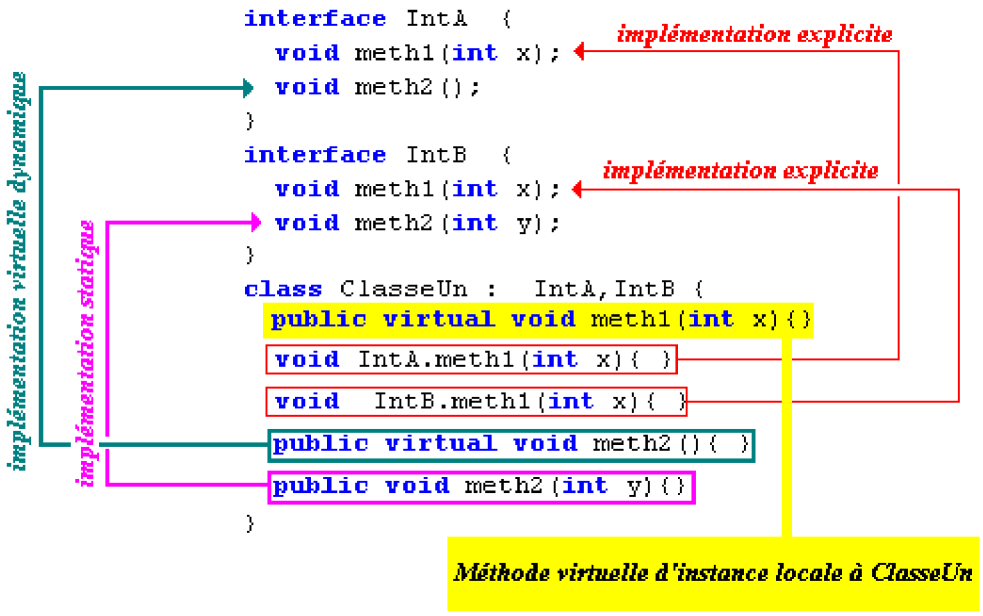
L'exemple ci-dessous est figuré avec deux interfaces IntA et IntB contenant chacune deux méthodes portant les mêmes noms et plus particulièrement la méthode meth1 possède la même signature dans chaque interface. Soit ClasseUn une classe implémentant ces deux interfaces. Voici comment fait C# pour choisir les appels de méthodes implantées.

Le code source de ClasseUn implémentant les deux interfaces IntA et IntB :

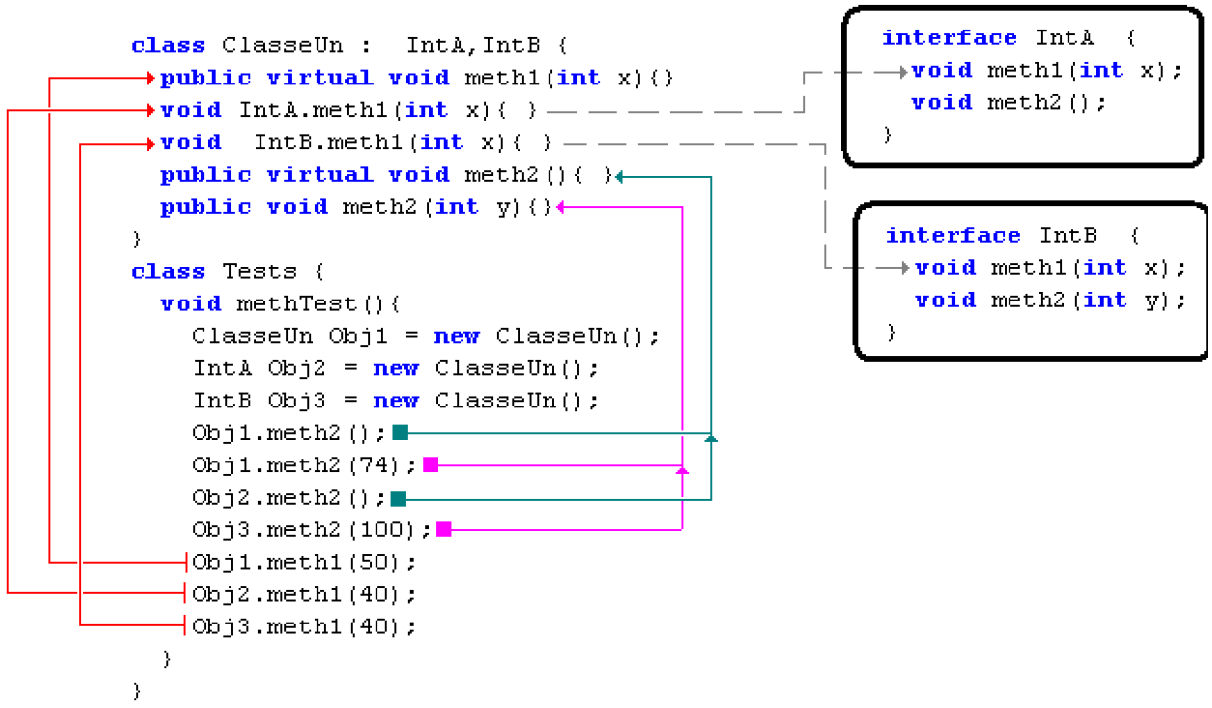
```
interface IntA{
 void meth1( int x );
 void meth2( );
}
interface IntB{
 void meth1( int x );
 void meth2( int y );
}
```

```
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.*

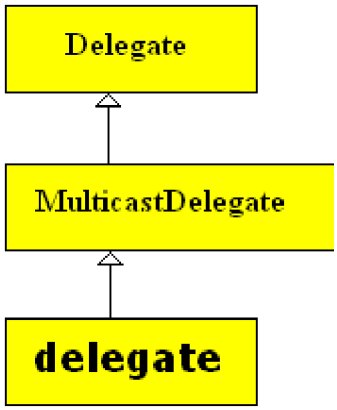
1.1 Définition classe de délégation - délégué

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 parlerons de la classe de délégation d'un objet délégué.

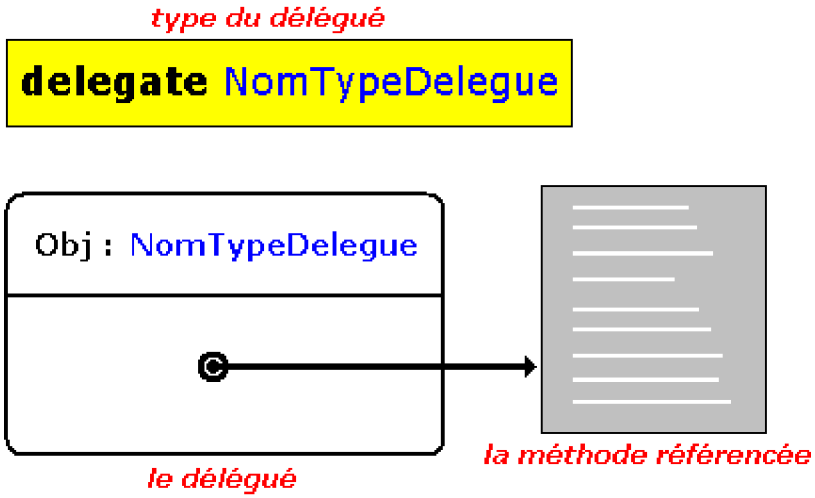
Il donc possible de créer un nouveau type de délégué (une nouvelle classe) dans un programme, ceci d'une seule manière : en utilisant le qualificateur **delegate**. Ci-dessous nous déclarons un type délégué (une nouvelle classe particulière) nommé NomTypeDelegue :

```
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) ;

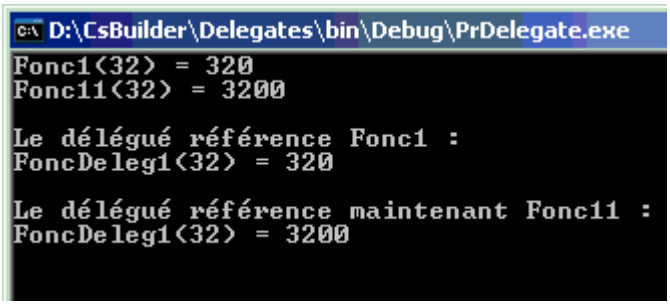
On peut maintenant appeler le délégué FoncDeleg1 dans une instruction avec un paramètre d'entrée de type **int**, selon que le délégué référence Fonc1 ou bien Fonc11 c'est l'une ou l'autre des ces fonctions qui est en fait appelée.

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 ( );
  }
}
```

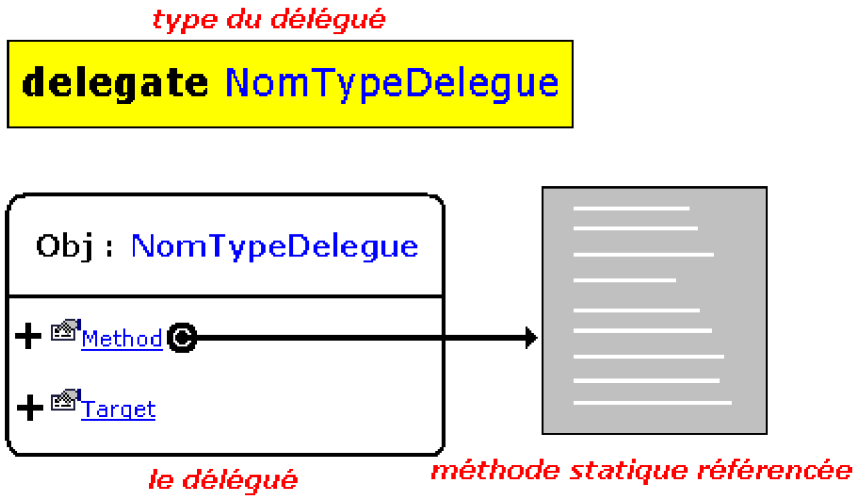
Résultats d'exécution sur la console :



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** n'est utilisable que 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**)

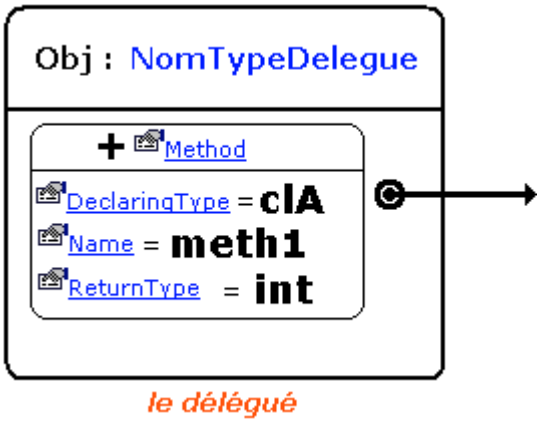
Obtient la méthode static représentée par le délégué.

```
[Serializable]
[ClassInterface(ClassInterfaceType.AutoDual)]
public MethodInfo Method {get;}
```

MethodInfo : Propriétés publiques

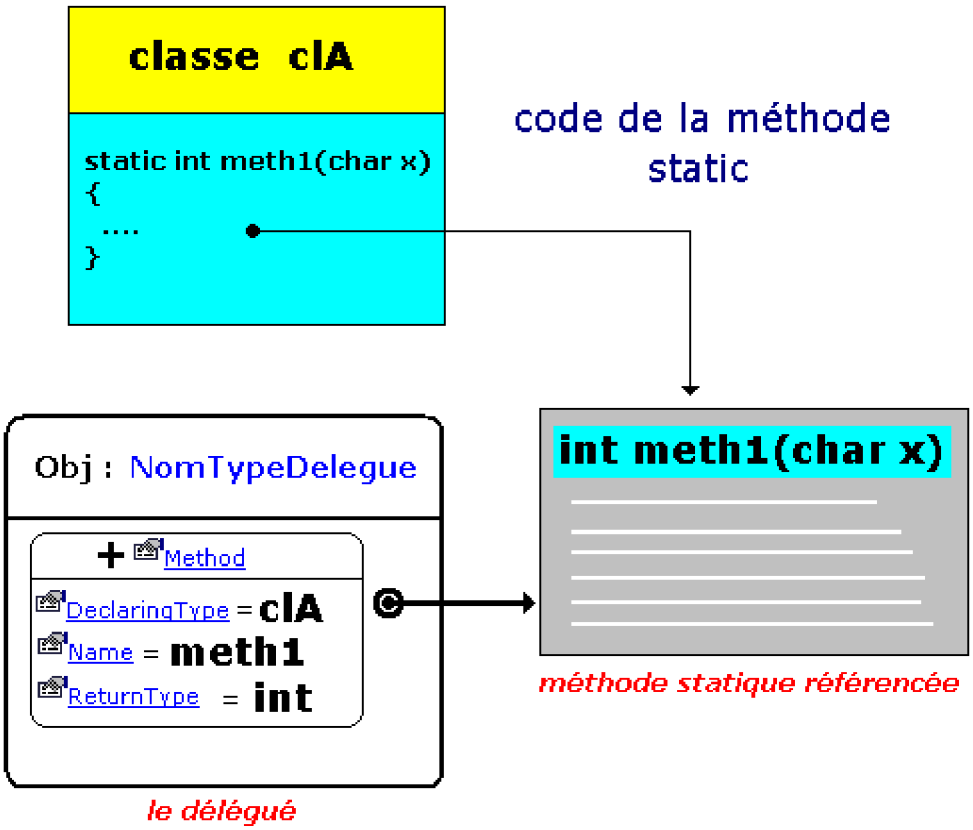
CallingConvention (hérité de **MethodBase**)	Obtient une valeur indiquant les conventions d'appel de cette méthode.
DeclaringType (hérité de **MemberInfo**)	Obtient la classe qui déclare ce membre.
Name (hérité de **MemberInfo**)	Obtient le nom de ce membre.
ReturnType	Obtient le type de retour de cette 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 un 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

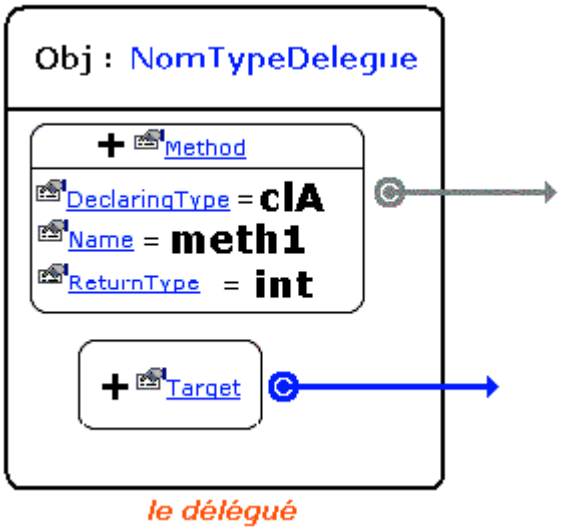
```
delegate int Deleguer( char x ) ;
```

3°) Instancier un objet de la classe clA

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

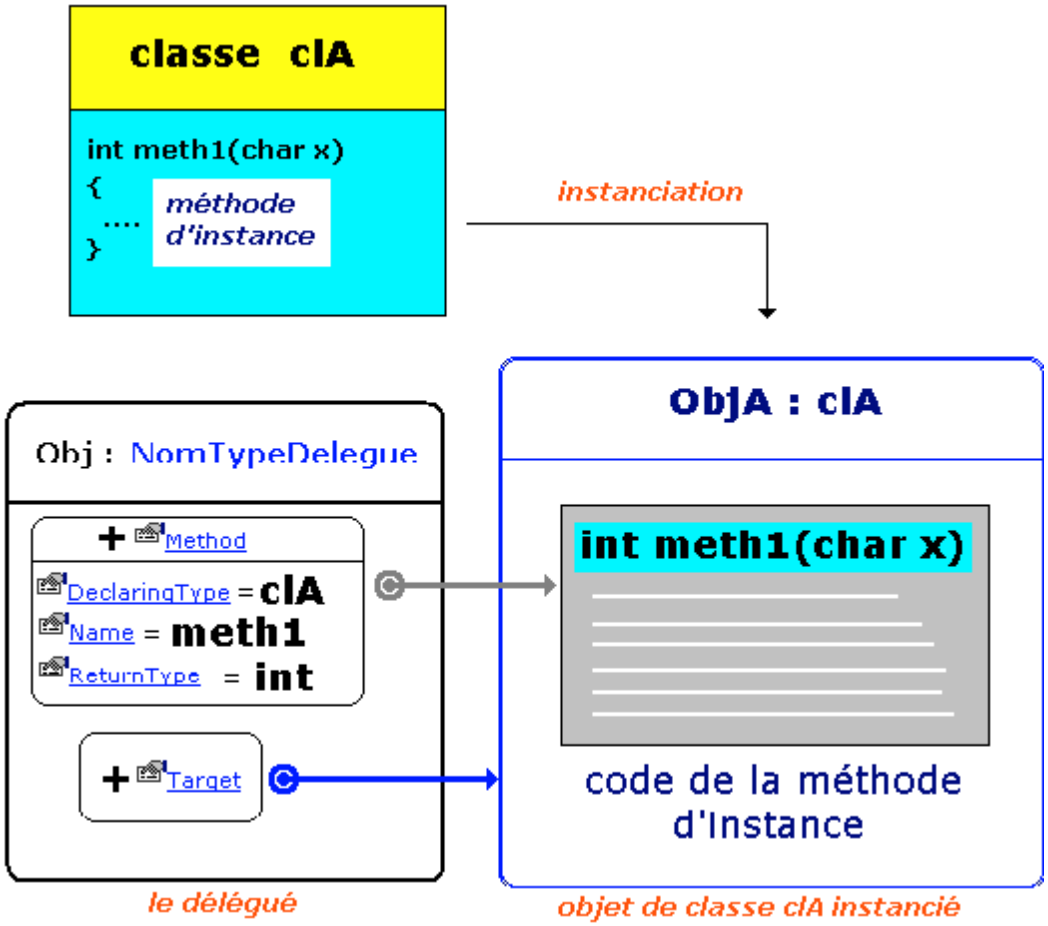
4°) Instancier à partir de la classe Deleguer un délégué

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



le délégué

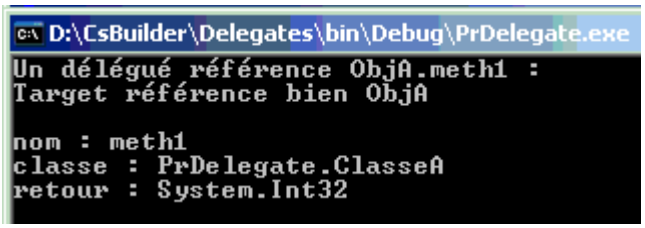
Nous illustrons dans la figure ci-après, dans le cas d'une méthode d'instance, l'utilisation de membres de la propriété **Method** et de la propriété **Target** :



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 :



Dans le programme précédent, les lignes de code suivantes :

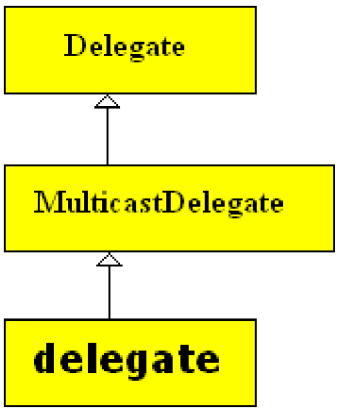
```
ObjX = (ClasseA)FoncDeleg.Target ;
if (ObjX.Equals(ObjA))
      System.Console.WriteLine ("Target référence bien ObjA") ;
else System.Console.WriteLine ("Target ne référence pas ObjA") ;
```

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 :

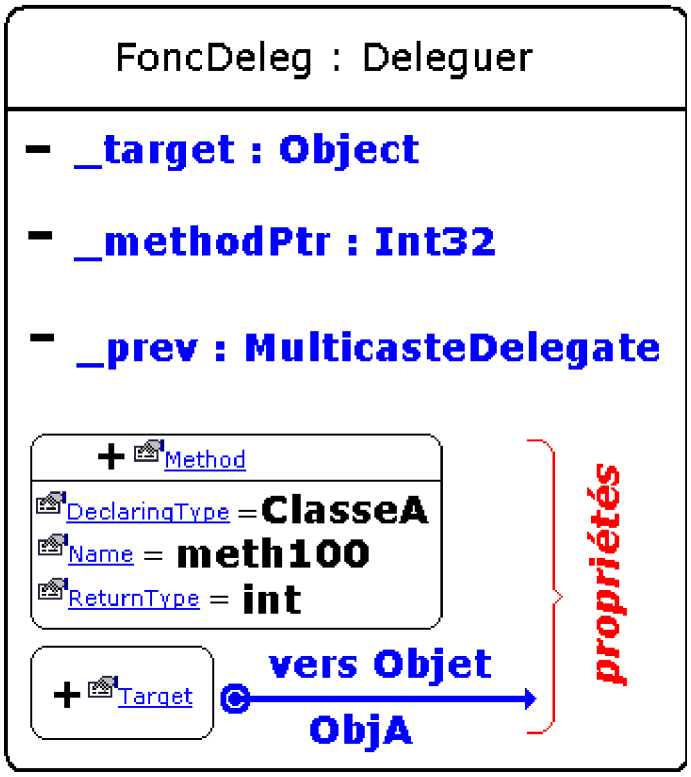
```
Object  _target ;
Int32  _methodPtr ;
```

MulticasteDelegate _prev ;
Le champ **_prev** est utilisé pour maintenir une liste de MulticasteDelegate

Lorsque nous déclarons un programme comme celui-ci :

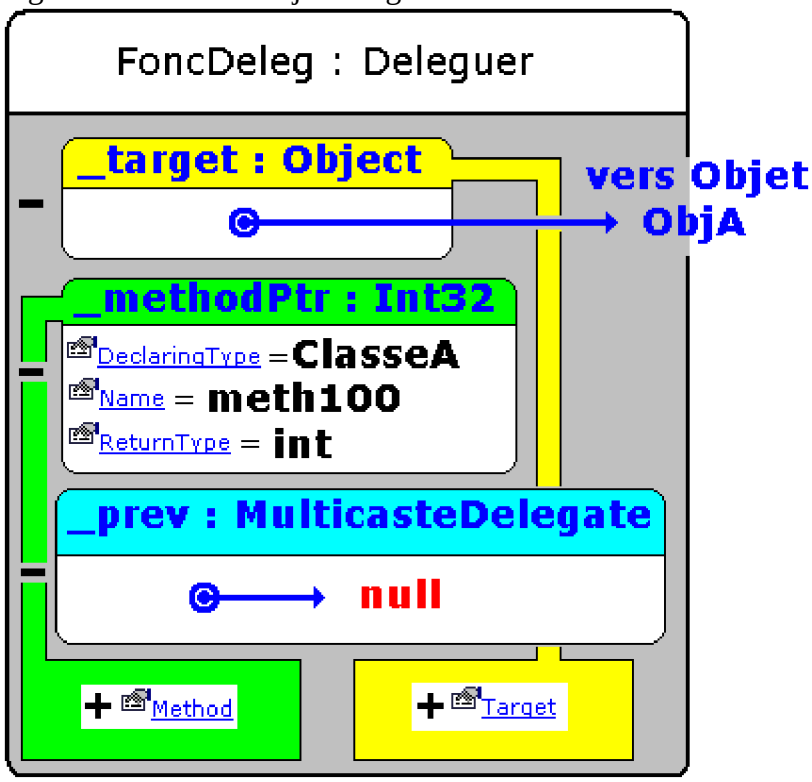
```
delegate  int  Deleguer ( char  x );

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.

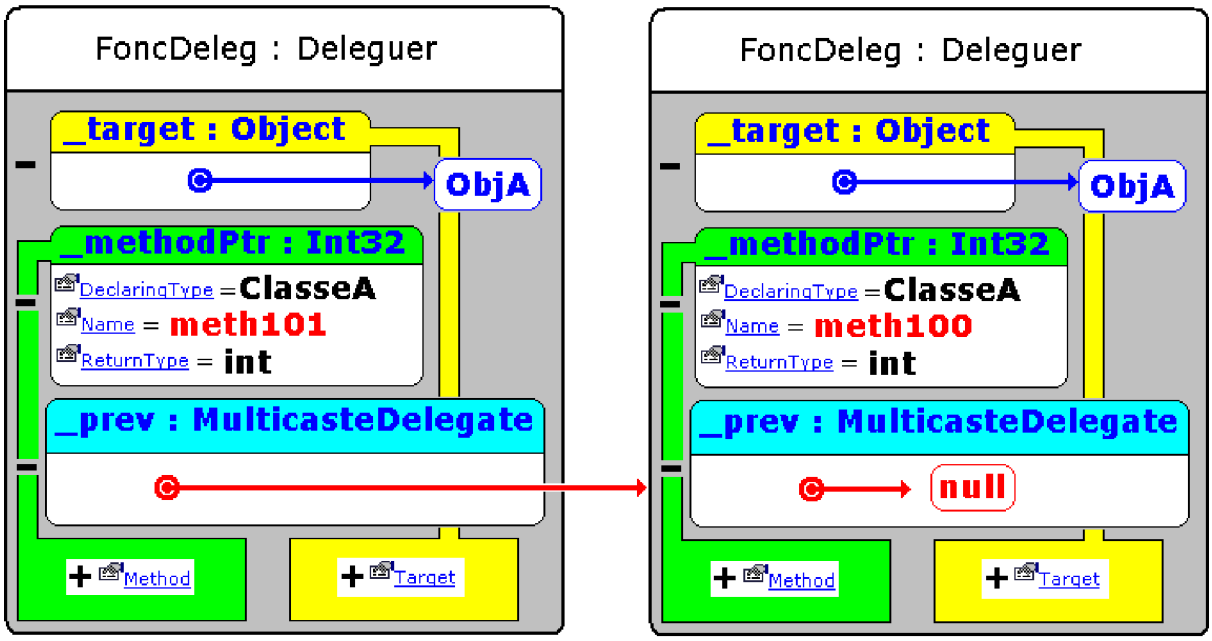
Figure virtuelle de l'objet délégué à ce stade :



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 :

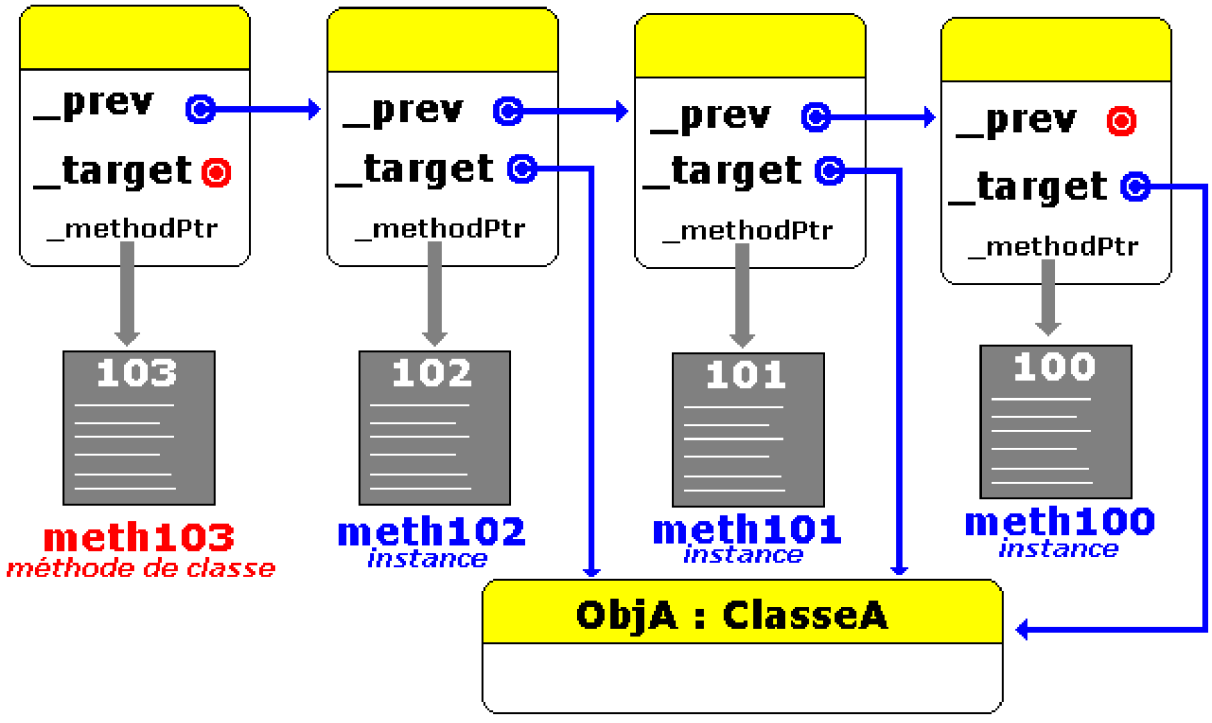
```
namespace PrDelegate {

 delegate  int  Deleguer ( char  x );

 class  ClasseA {

   public int 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 référencement 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
----------------------------------
```

IHM et Winforms

- Les événements
- Propriétés et indexeurs
- Fenêtres et ressources
- Contrôles dans les formulaires
- 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

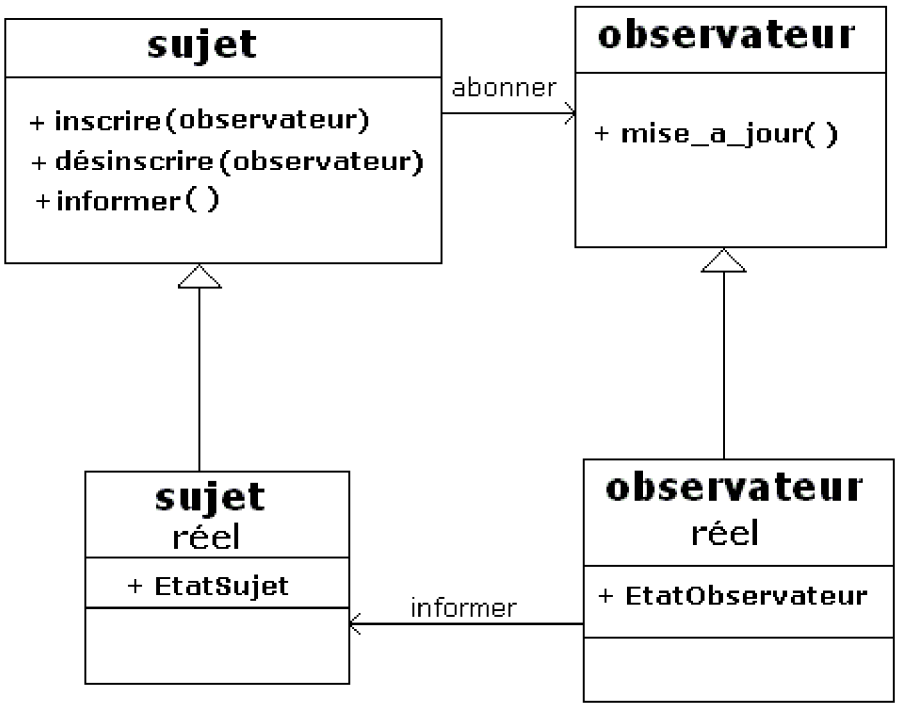
Rappel

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

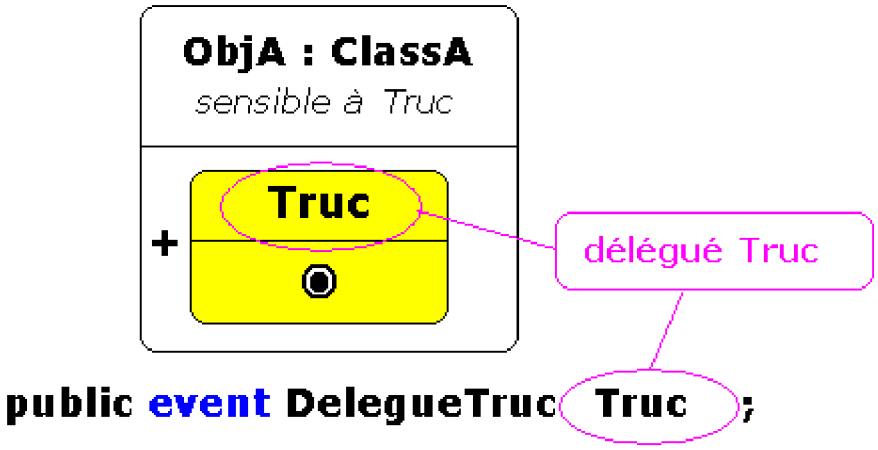
C# utilise les délégués pour fournir un mécanisme explicite permettant de gérer l'abonnement/notification.

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

type délégation :

```
public Delegate void DelegueTruc( ... ) ;
```



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).

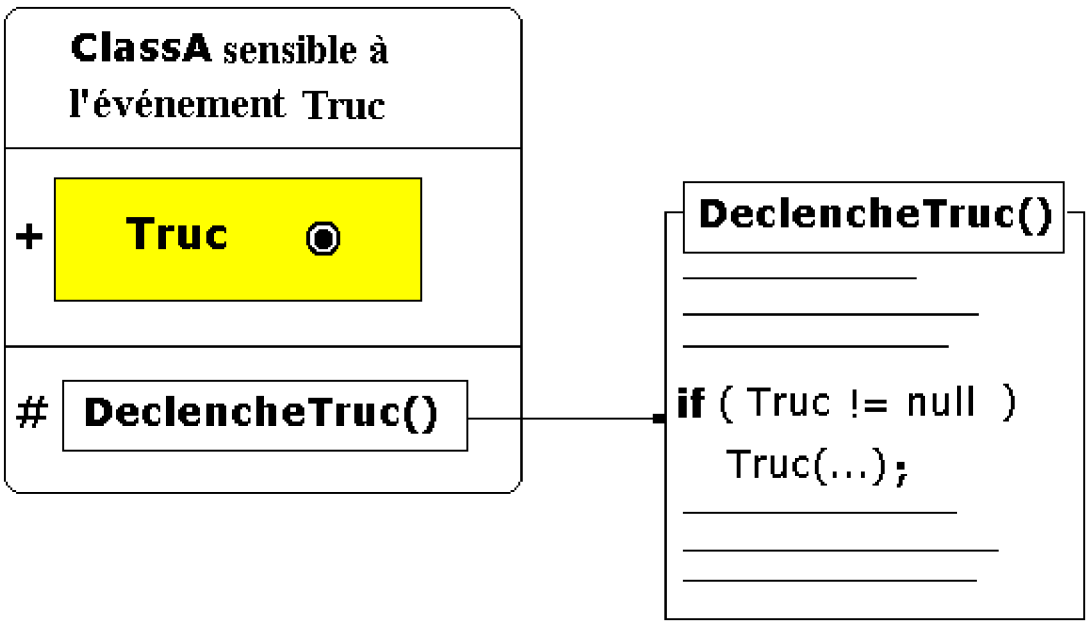
Remarque importante

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).

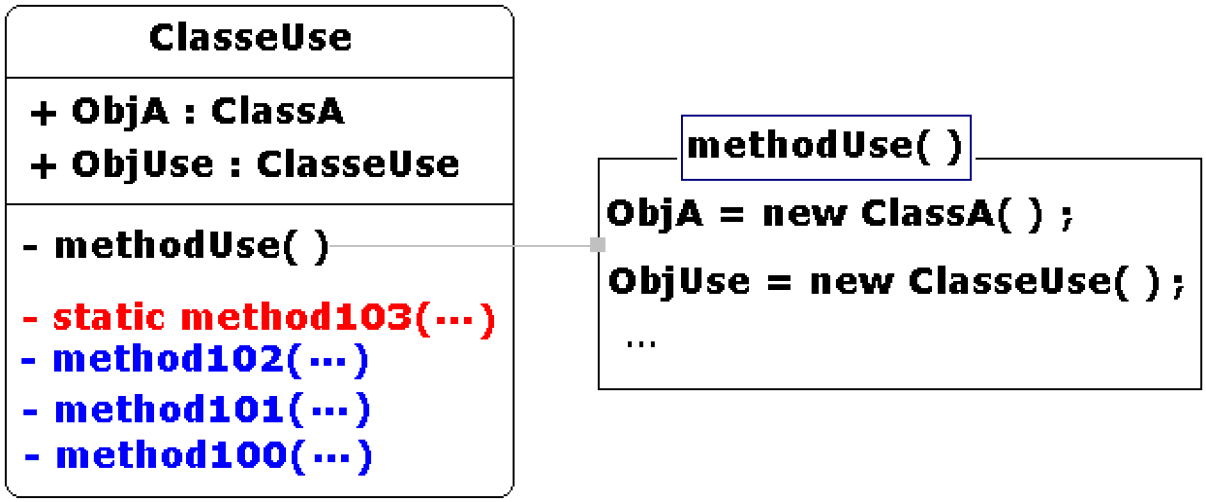
Enrichissons le code C# précédent avec la classe ClasseUse :

```
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 :

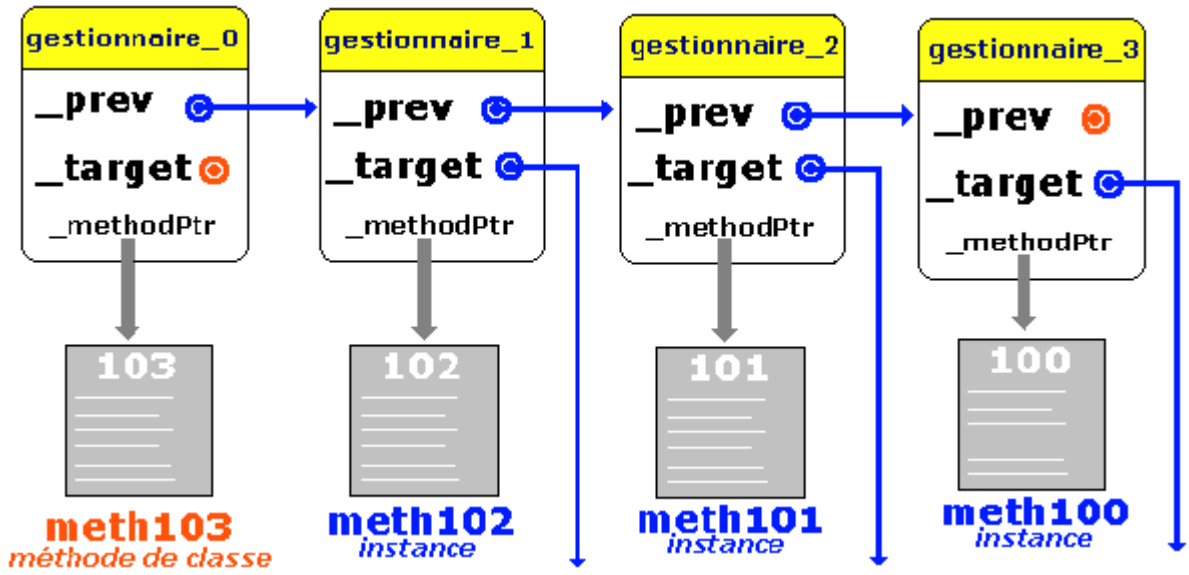


Ajoutons au code C# de la classe ClassA les quatre gestionnaires :

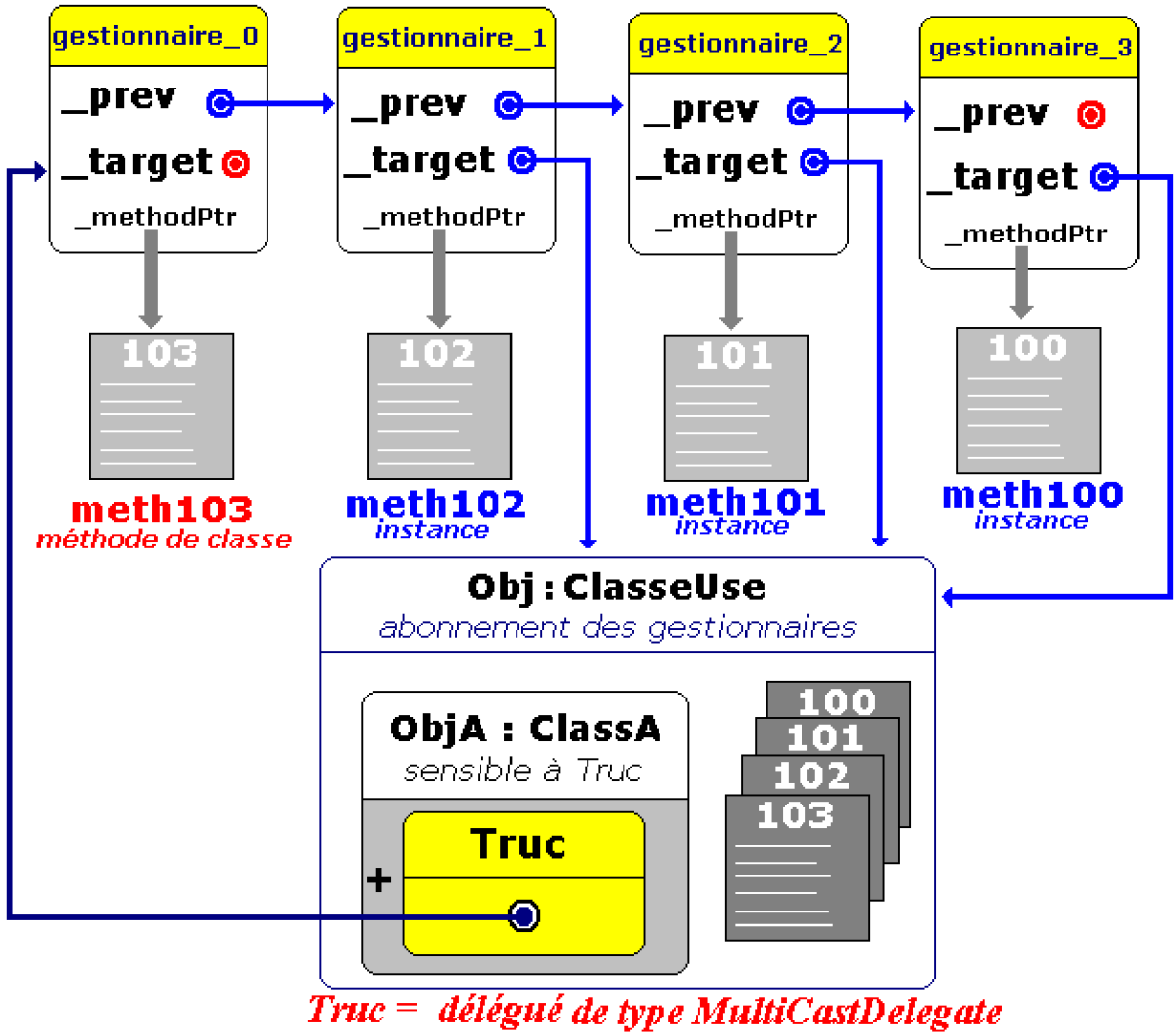
```
using System;
using System.Collections;

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.DeclencheTruc( ) ;  //...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 :

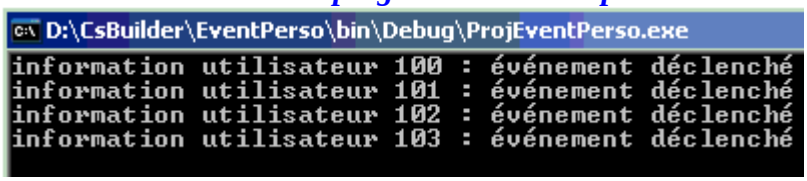
```
        public void LancerTruc( ) {
          //....
          TrucEventArgs evt = new TrucEventArgs ("événement déclenché" ) ;
          OnTruc ( this , evt );
          //....
        }

    }

    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 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 du programme console précédent :



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

Empty

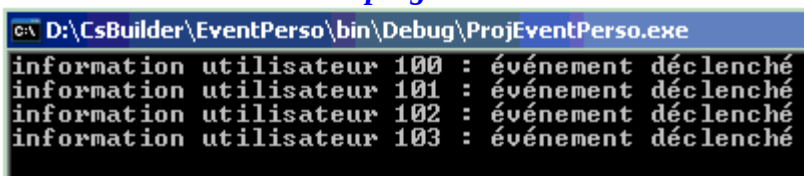
Représente un événement sans données d'événement.

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.

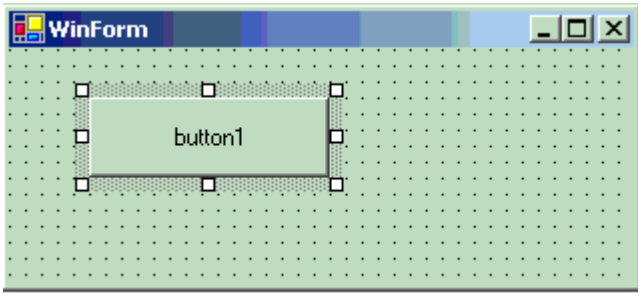
Ces 58 événements sont tous normalisés, certains sont des événements sans information spécifique, d'autres possèdent des informations spécifiques, ci-dessous un extrait de la liste des événements de la classe Control, plus particulièrement les événements traitant des actions de souris :

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 un 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.

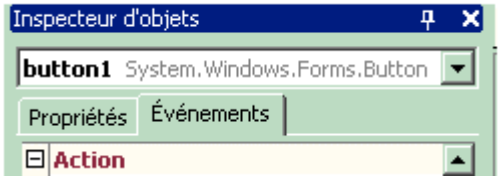
Il nous reste les étapes suivantes :

- 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)

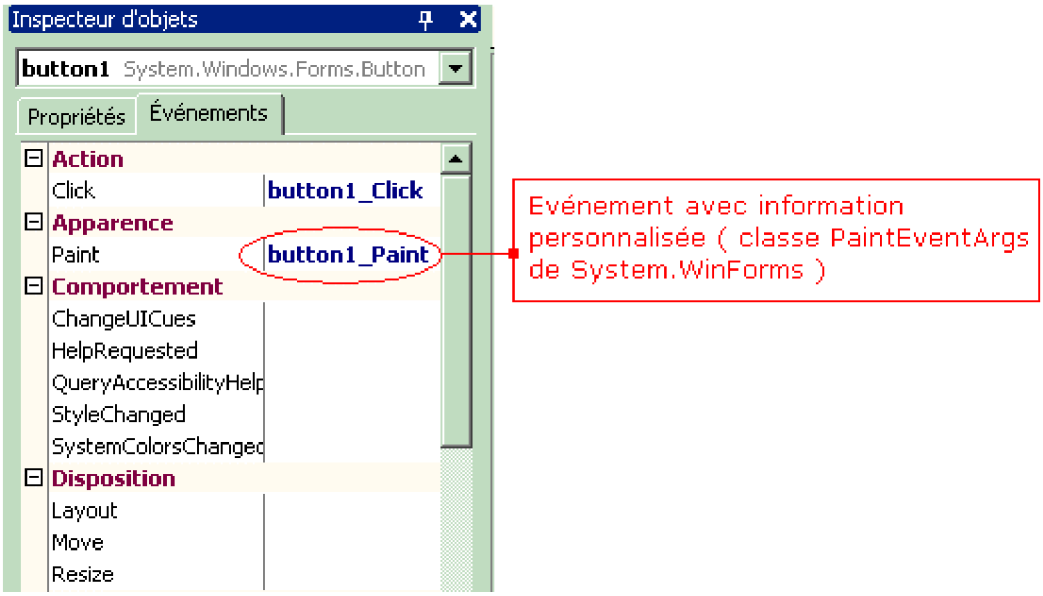
L'étape 7° est assurée par le système d'exploitation qui se charge d'envoyer des messages et de lancer les événements.

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.

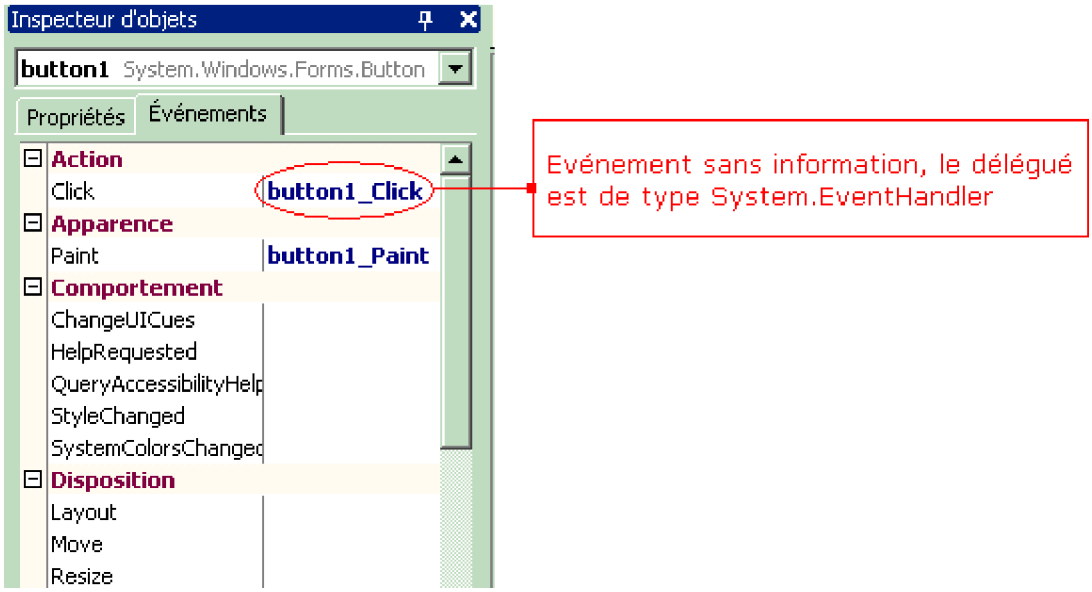
Méthodes publiques

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 C#Builder comme dans Delphi, 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

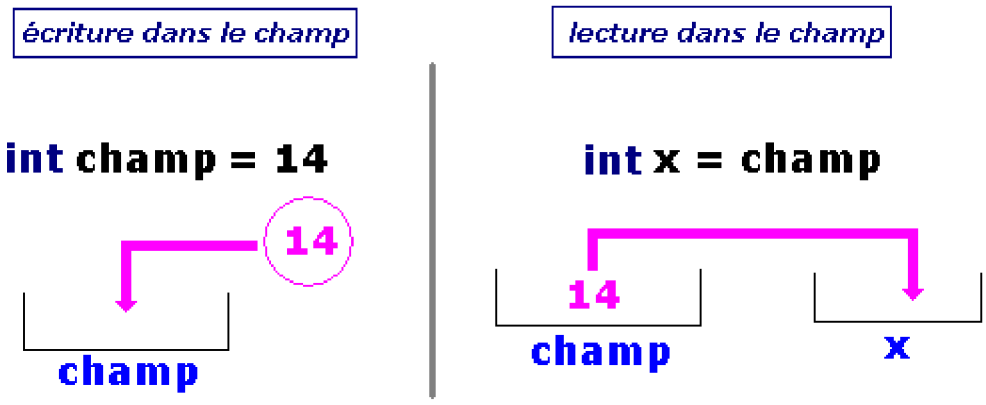
Définition d'une propriété

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** :

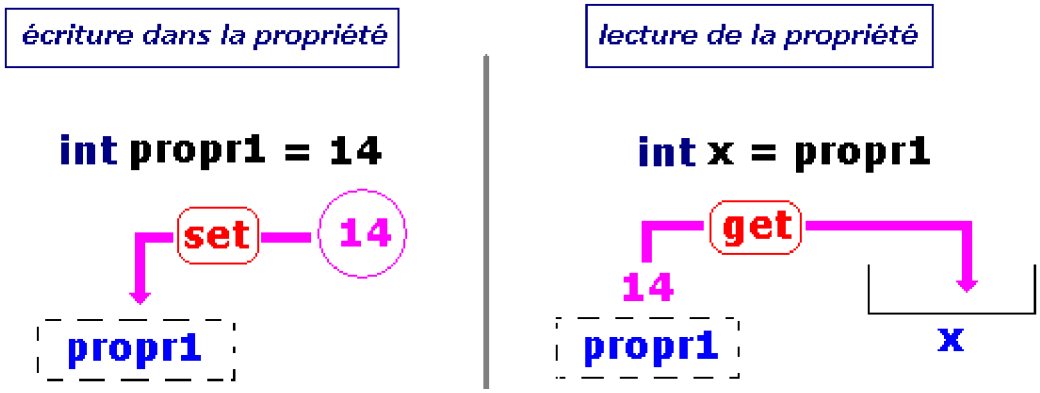
```
public int propr1 {
        //..... bloc de définition
}
```

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  ; }
}
```

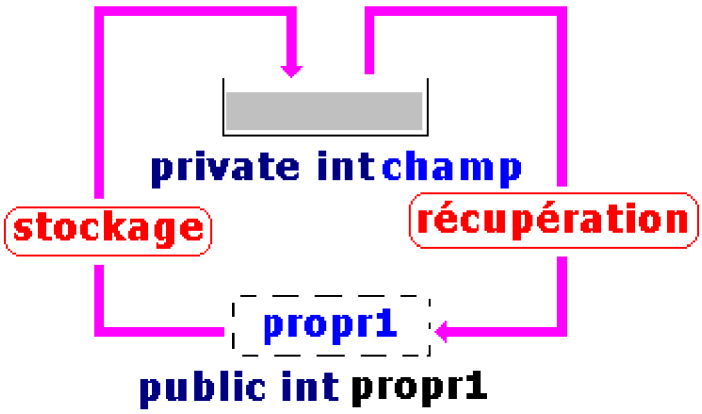
Le mot clef ***value*** est une sorte de paramètre implicite interne à l'accesseur **set**, il contient la valeur effective qui est transmise à la propriété lors de l'accès en écriture.

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 accéde 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 " :

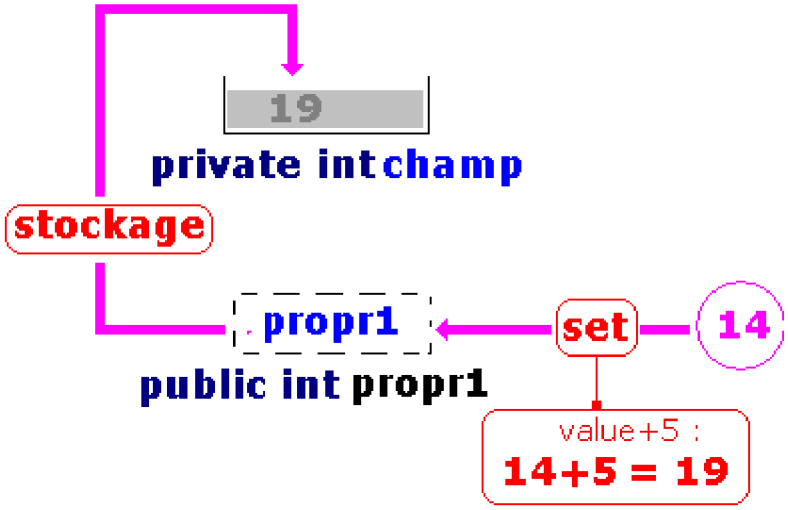
```
private int champ;

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.*

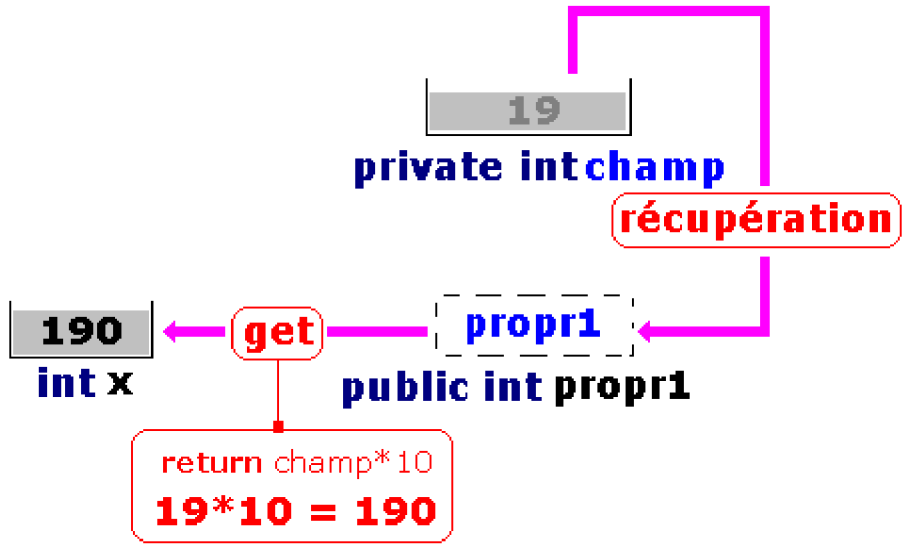


Utilisons maintenant notre propriété en mode lecture à travers une affectation :

```
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

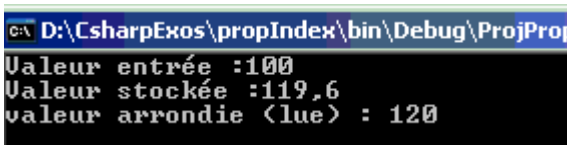
Une propriété servant à fournir automatiquement le prix d'un article en y intégrant la TVA au taux de 19.6% et arrondi à l'unité d'euro supérieur :

```
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 :



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

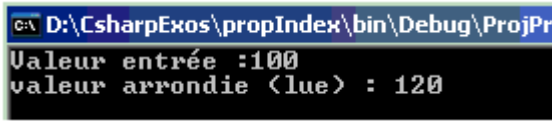
Les propriétés, comme les champs peuvent être des **propriétés de classes** et donc qualifiées par les mots clefs comme **static**, **abstract** etc ...Dans l'exemple précédent nous avons qualifié tous les champs et la propriété prix en **static** afin qu'ils puissent être accessibles à la méthode **Main** 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 :



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 :

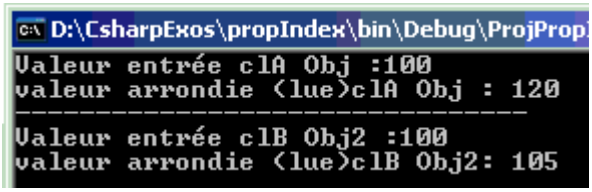
[C# Avertissement] Class..... : Le mot clé new est requis sur '.....', car il masque le membre hérité..... '

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 :



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   { // implantation par masquage de la propriété de la classe
mère
```

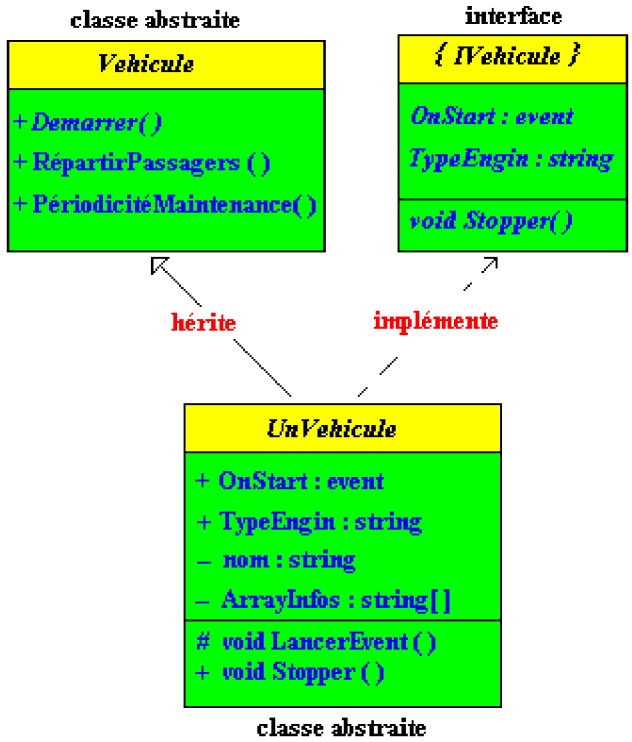
```
    get    { return  Math.Round ( prixLocal ) ; }
    set    {    prixTotal  =  value  *  tauxTVA  ;  }
  }
 }

class  clB2  :  clA   {
   private  Double prixTotal  ;
   private  Double tauxTVA  =  1.05  ;
  public override  Double prix    { // implantation par redéfinition de la propriété de la
classe mère
    get    { return  Math.Round ( prixLocal ) ; }
    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 {
  private  string  nomTerre = "";
  ....
  public override  string  TypeEngin     { // redéfinition de propriété
   get   {   return  base .TypeEngin ;  }
   set   {   string  nomTerre = value + "-Terrestre";
             base .TypeEngin = nomTerre ;  }
  }
}
```

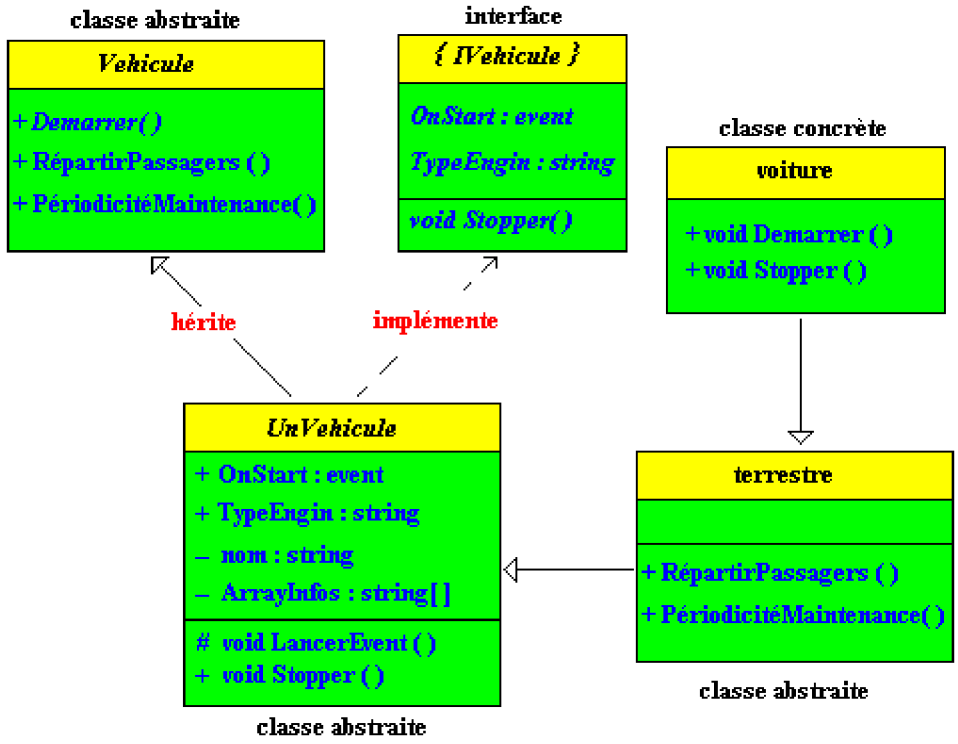
1.9 Exemple complet exécutable

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
```

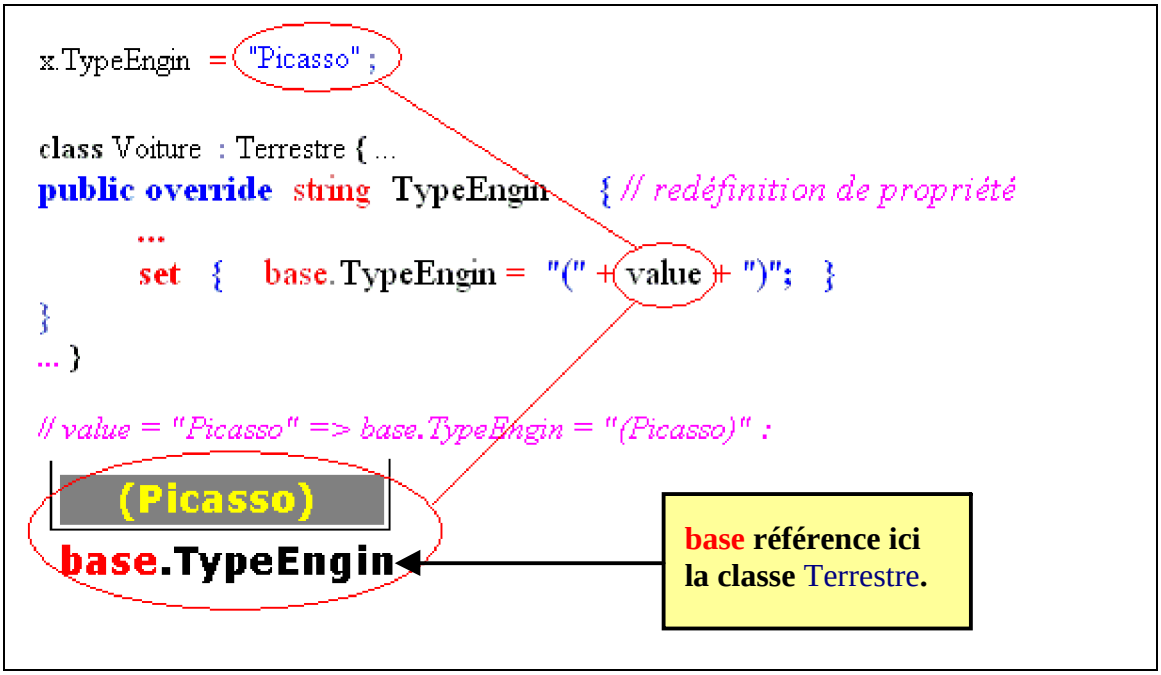
- **propriété TypeEngin dans la classe Terrestre :**

```
(Picasso)-Terrestre
.....   base.TypeEngin
```

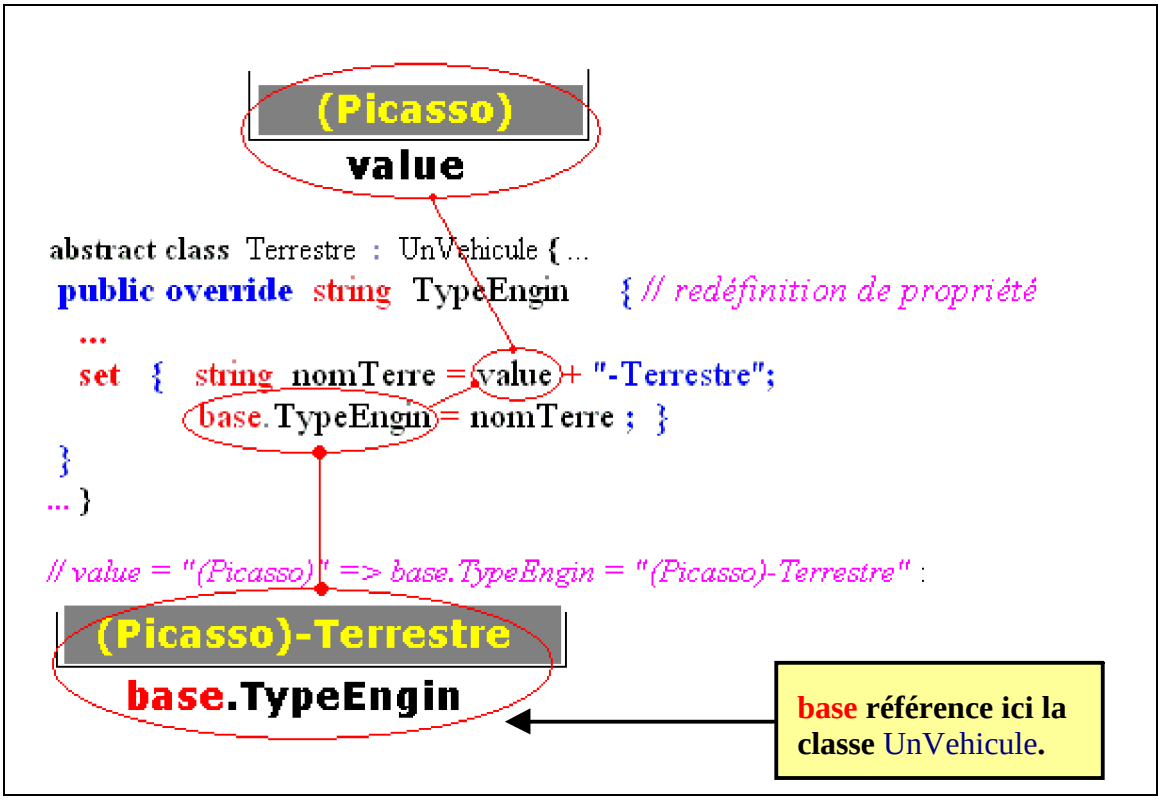
- **propriété TypeEngin dans la classe UnVehicule :**

```
[(Picasso)-Terrestre]
..................   private string nom
```

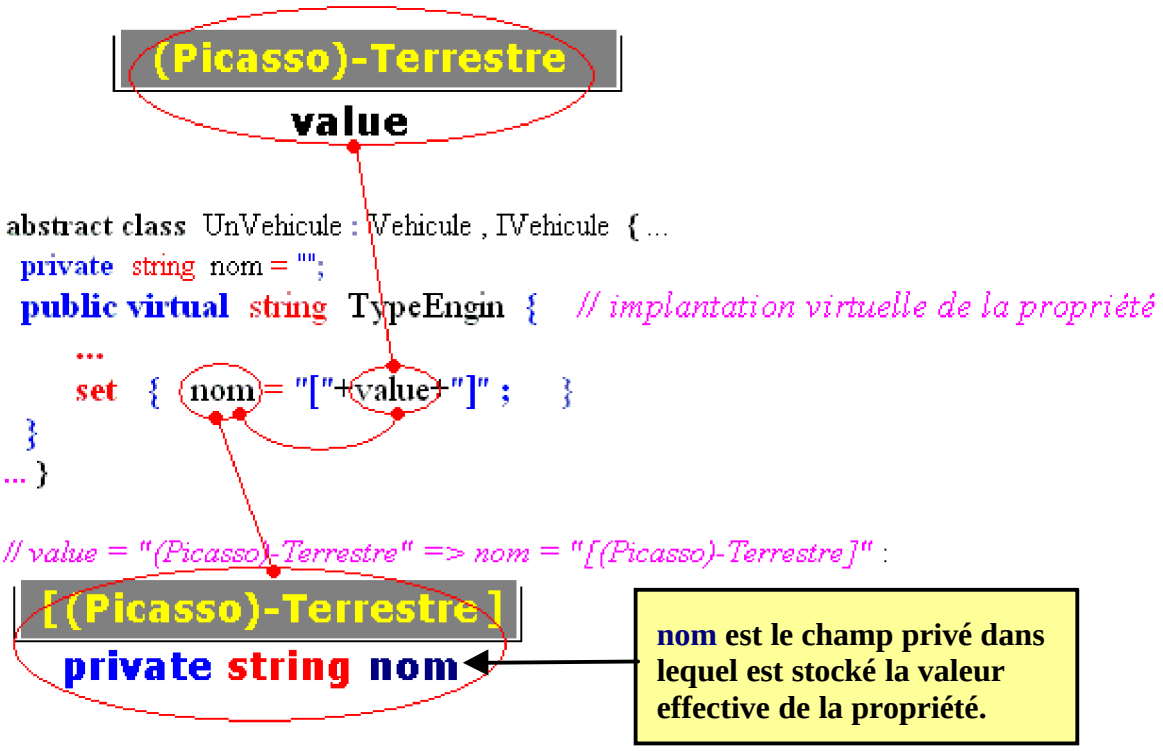
Définition de la propriété dans la classe Voiture (écriture) :



Définition de la propriété dans la classe Terrestre (écriture) :



Définition de la propriété dans la classe UnVehicule (écriture) :

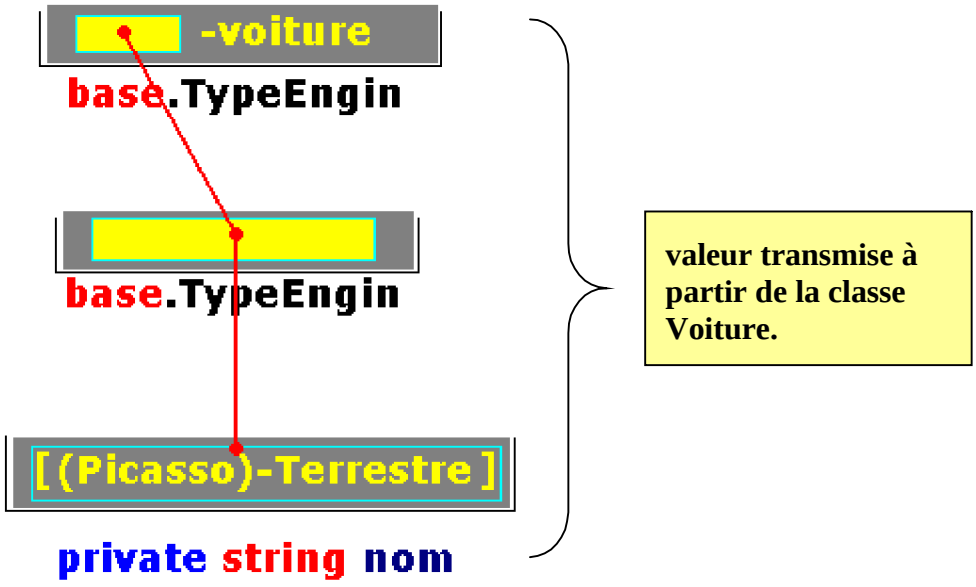


1.9.2 Détails de fonctionnement de la propriété TypeEngin en lecture

La propriété TypeEngin est en lecure dans :

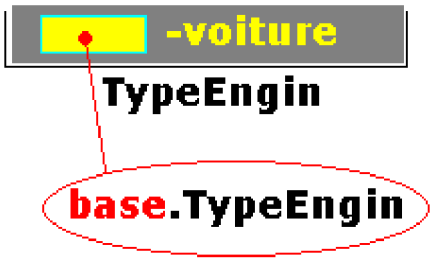
```
System.Console.WriteLine ( "x est une " + x.TypeEngin ) ;
```

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) :

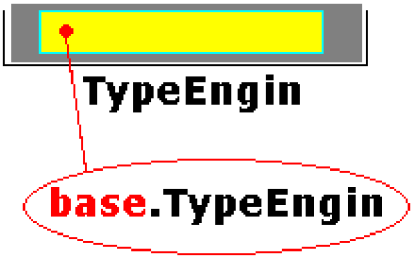
```
class Voiture  : Terrestre { ...
  public override  string  TypeEngin     { // redéfinition de propriété
        get   {   return  base.TypeEngin + "-voiture";  }
       ...
  }
... }
```



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) :

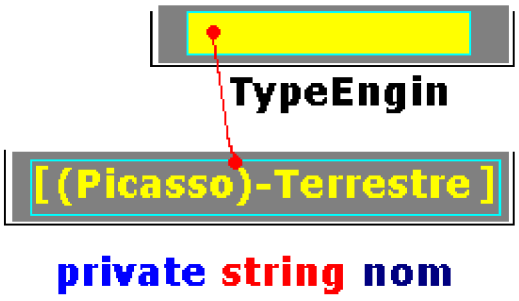
```
abstract class  Terrestre  :  UnVehicule { ...
  public override  string  TypeEngin     { // redéfinition de propriété
     get   {   return  base.TypeEngin ;  }
     ...
  }
... }
```



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.

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

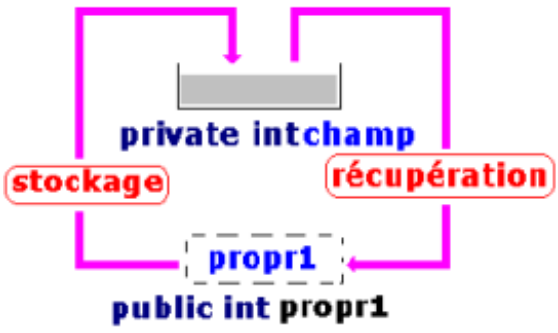
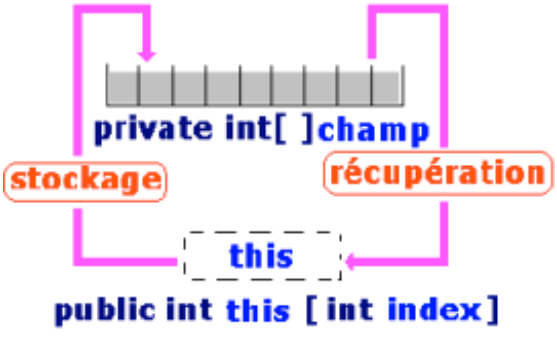
Un indexeur est un membre de classe qui permet à un objet d'être **indexé de la même manière qu'un tableau.** La signature d'un indexeur doit être différente des signatures de tous les autres indexeurs déclarés dans la même classe. Les indexeurs et les propriétés sont très similaires de par leur concept, 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.

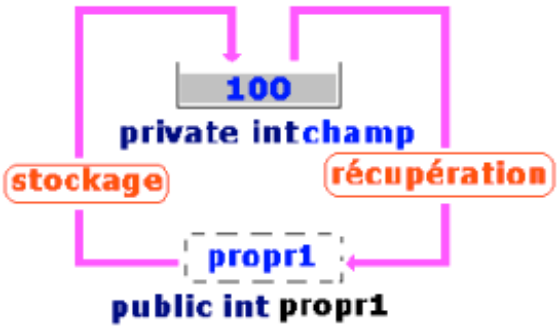
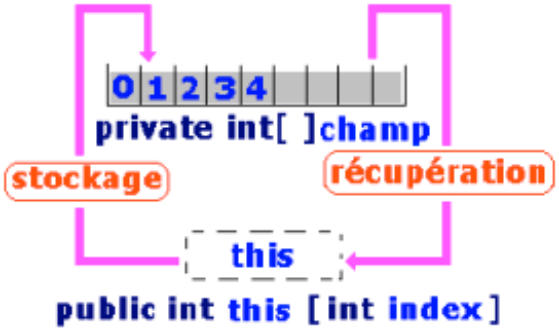
Propriété	Indexeur
Déclarée par son nom.	Déclaré par le mot clef **this**.
Identifiée et utilisée par son nom.	Identifié par sa signature, utilisé par l'opérateur **[ ]** . L'opérateur **[ ]** doit être situé immédiatement après le nom de l'objet.
Peut être un membre de classe **static** ou un membre 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*** ; **}** **}**  	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*** ; **}** **}** 

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( ); **Obj.propr1 = 100 ;**  **int** x = Obj.prop1 ; *// x = 100*	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( ); **for** (**int** i =0; i<5; i++) **Obj[i] = i ;**  **int** x = Obj[2] ; *// x = 2* **int** y = Obj[3] ; *// x = 3* ...

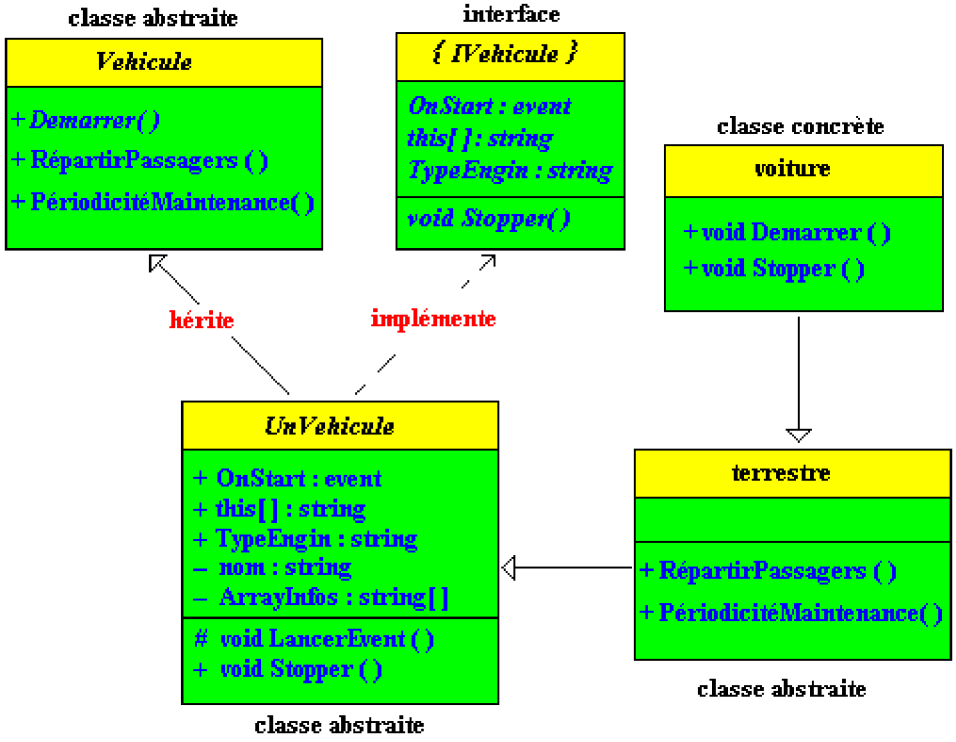
2.1.3 Paramètres

Propriété	Indexeur
Déclaration : **class** clA **{** **private int** champ; **public int** propr1**{** **get { return** champ*10 ; **}** **set {** champ = ***value*** + ***1*** ; **}** **}** **}** ***value*** est un paramètre implicite.	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( ); **Obj.propr1 = 100 ;** **int** x = Obj.prop1 ; *// x = 1010*	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:

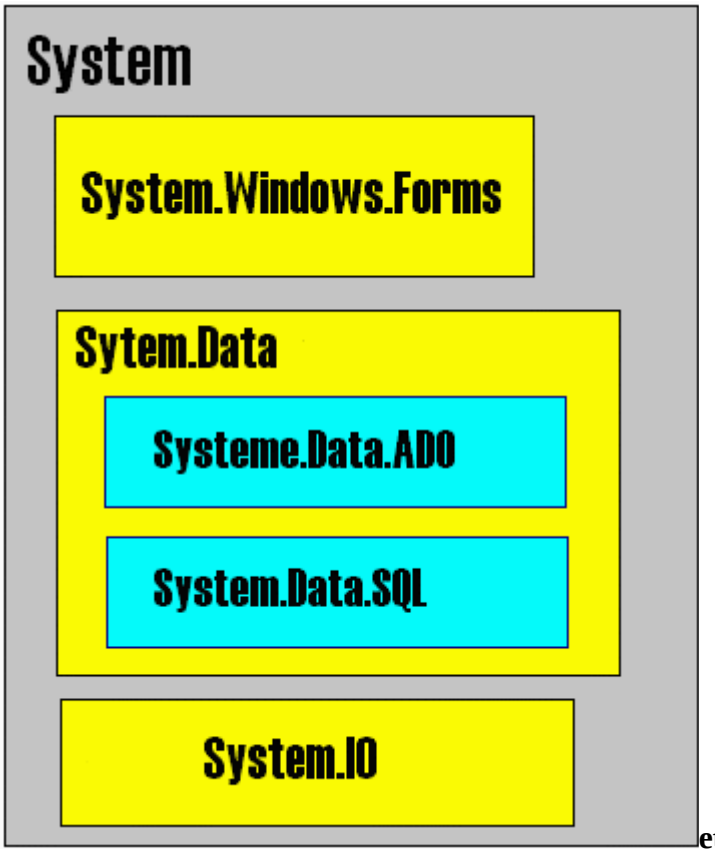
1. Les applications avec Interface Homme-Machine

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 avec NetFramWork 1.1., les paragraphes 1.3, 1.4, ... , 1.10 expliquent le contenu du code généré automatiquement par Visual Studio ou C# Builder 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.

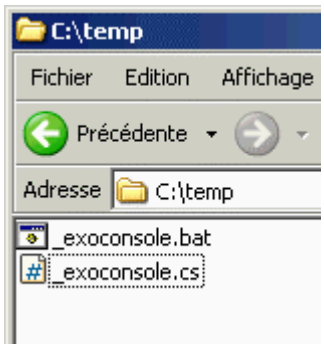
La classe **Form** de l'espace des noms **System.Windows.Forms** permet de créer une fenêtre classique avec barre de titre, zone client, boutons de fermeture, de zoom...

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

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#.

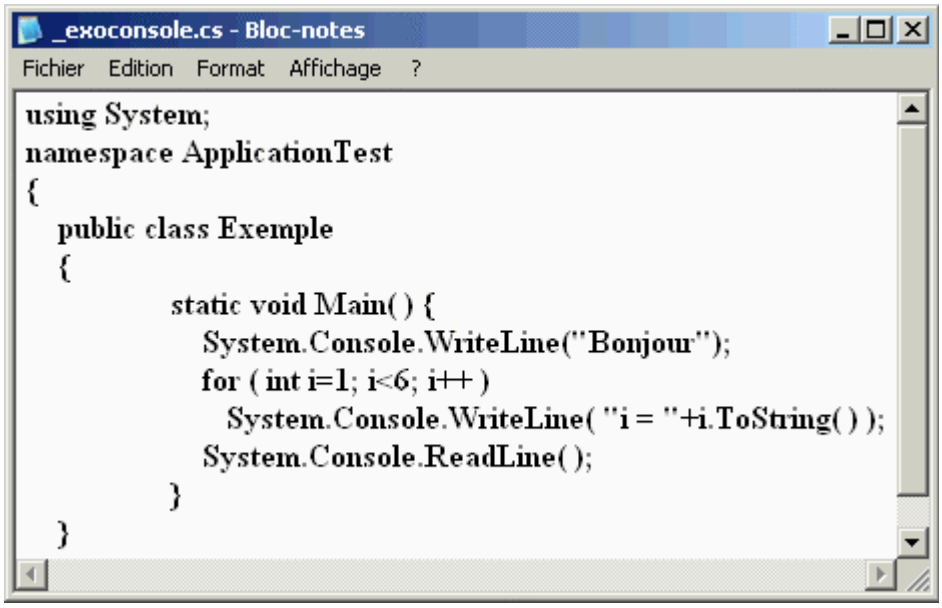
Construction de la commande de compilation "**_exoconsole.bat**" :

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)

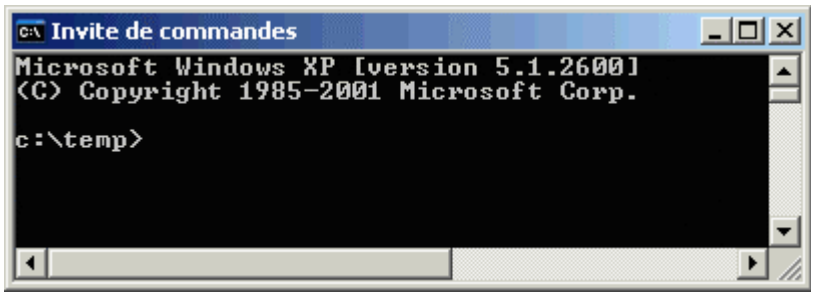
Texte de la commande dans le Bloc-note :

Nous donnons le contenu du fichier source **_exoconsole.cs** à compiler :

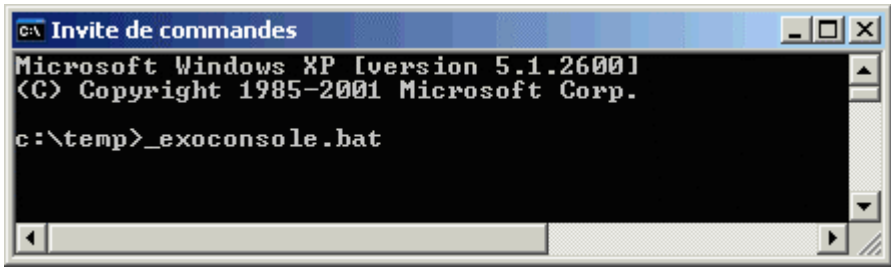


Le programme précédent affiche le mot Bonjour suivit de l'exécution de la boucle sur 5 itérations.

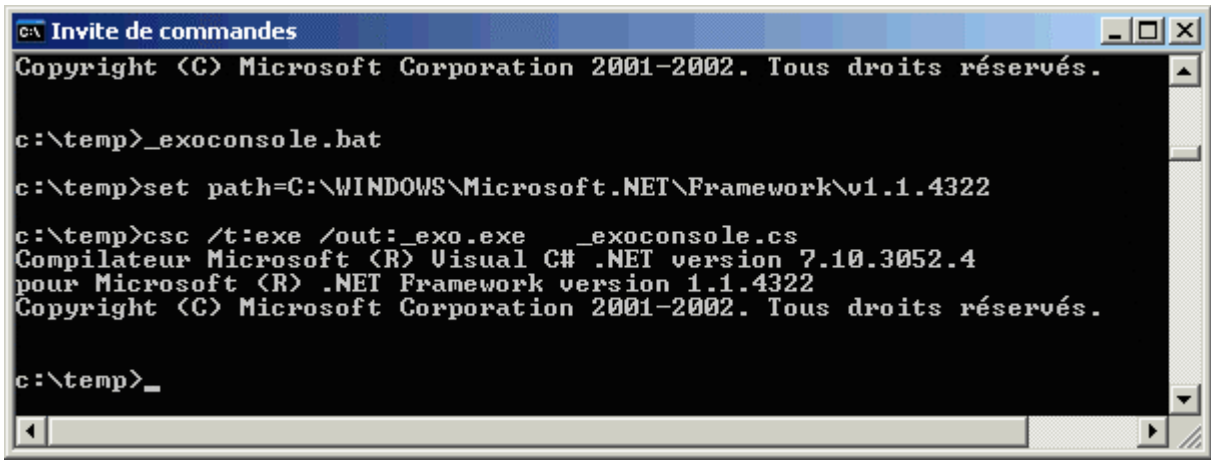
On lance l'invite de commande de Windows ici dans le répertoire c:\temp :



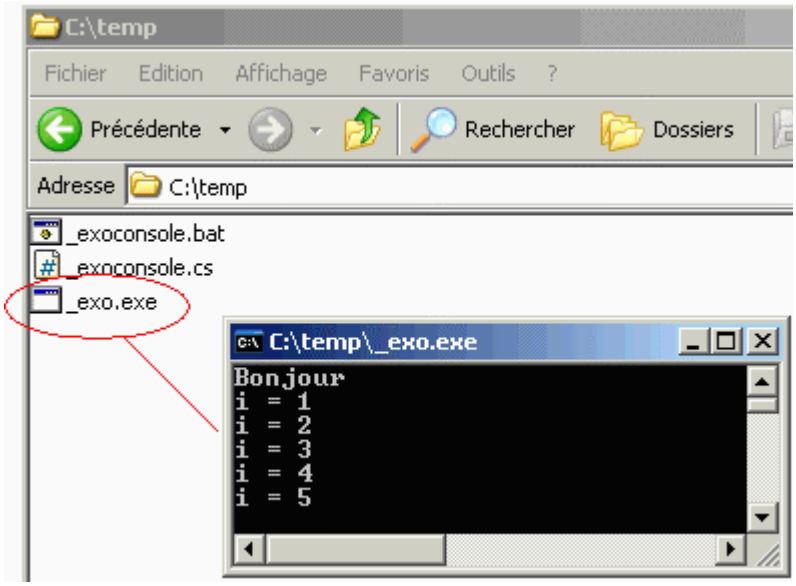
On tape au clavier et l'on exécute la commande "**_exoconsole.bat**" :



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** :



Lançons par un double click l'exécution du programme **_exo.exe** qui vient d'être engendré :

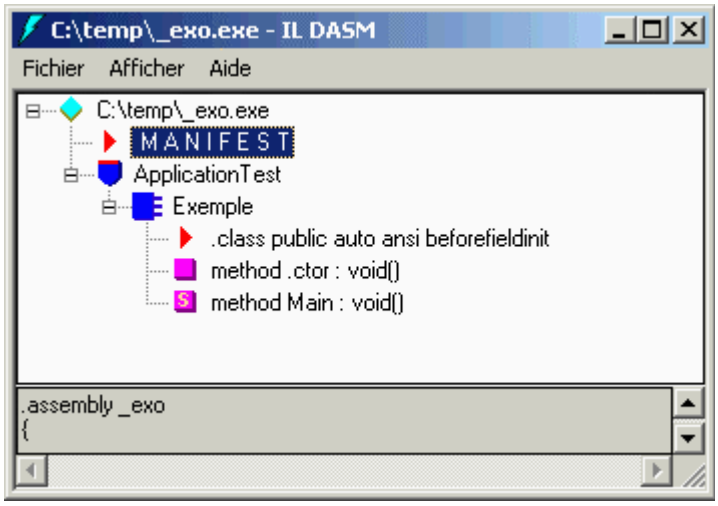


Nous constatons que l'exécution par le CLR du fichier **_exo.exe** a produit le résultat escompté c'est à dire l'affichage du mot Bonjour suivit de l'exécution de la boucle sur 5 itérations.

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
```

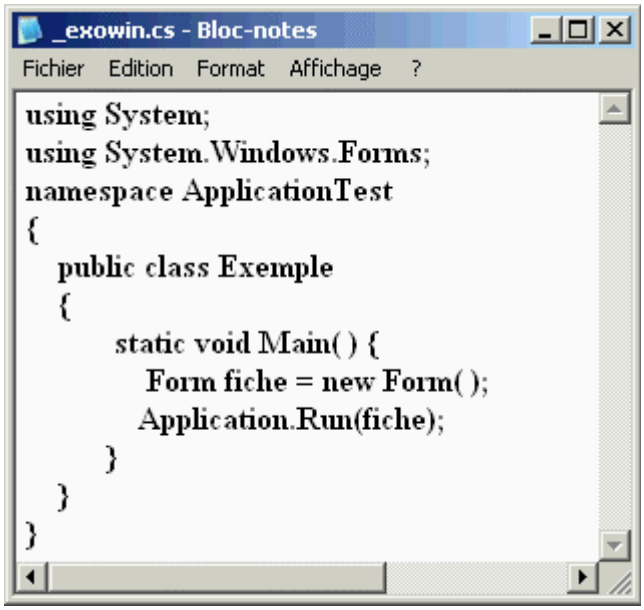
```
  //000011:    }
    IL_0032:  ret
  } // end of method Exemple::Main
```

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).

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( ) ;

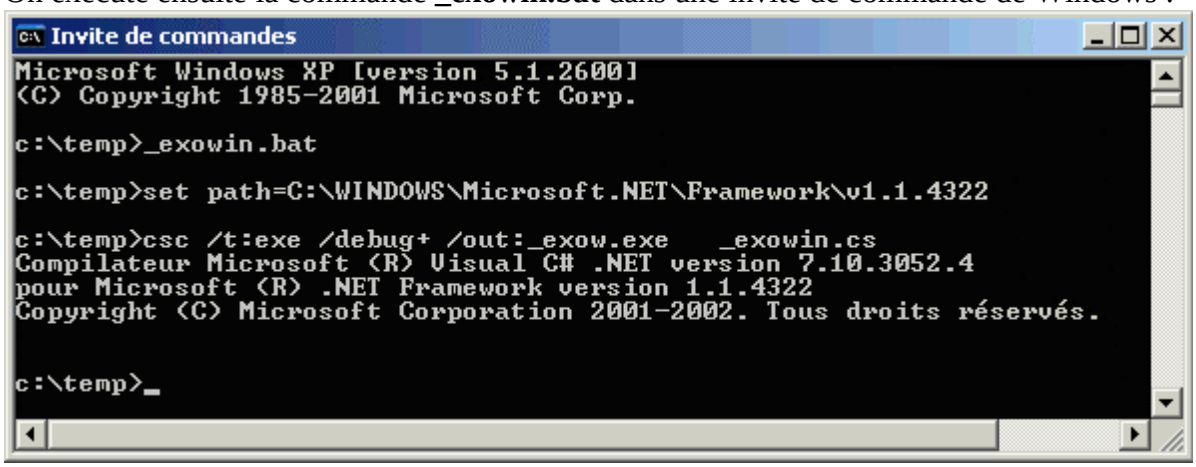
La fenêtre **fiche** est ensuite "initialisée" et "lancée" par l'instruction
Application.Run(fiche) ;

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.

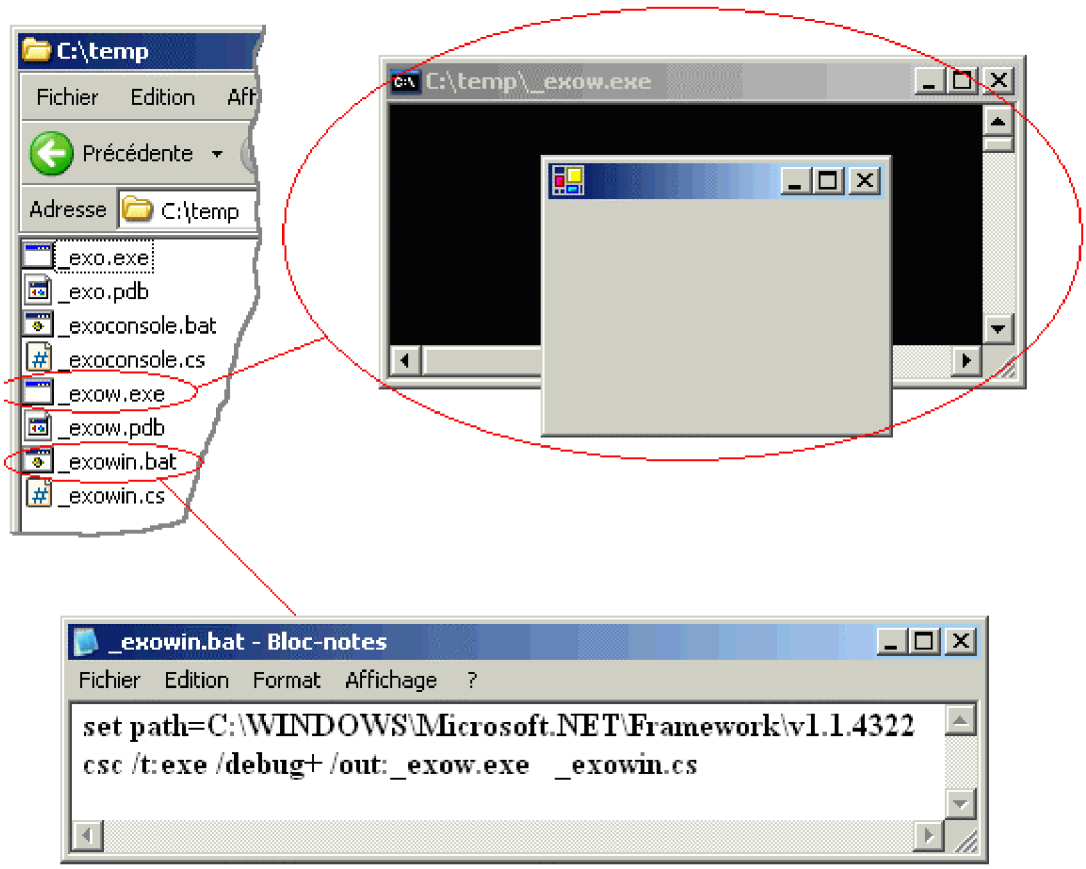
contenu fichier de commande **_exowin.bat** :

```
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.

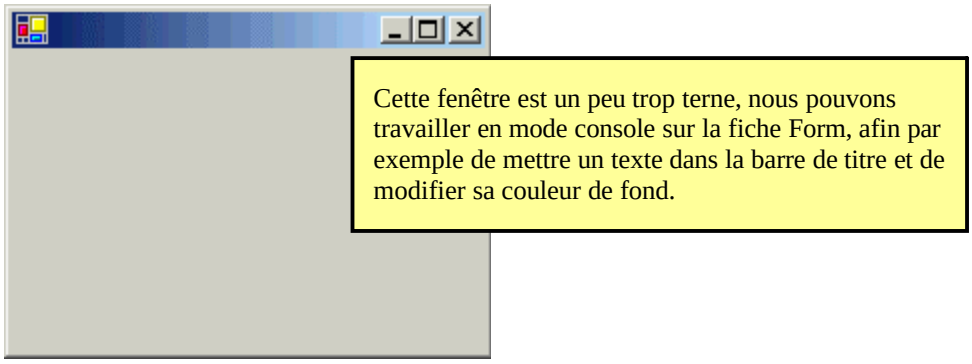
1.2.2 fenêtre personnalisée sans fenêtre console

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** :

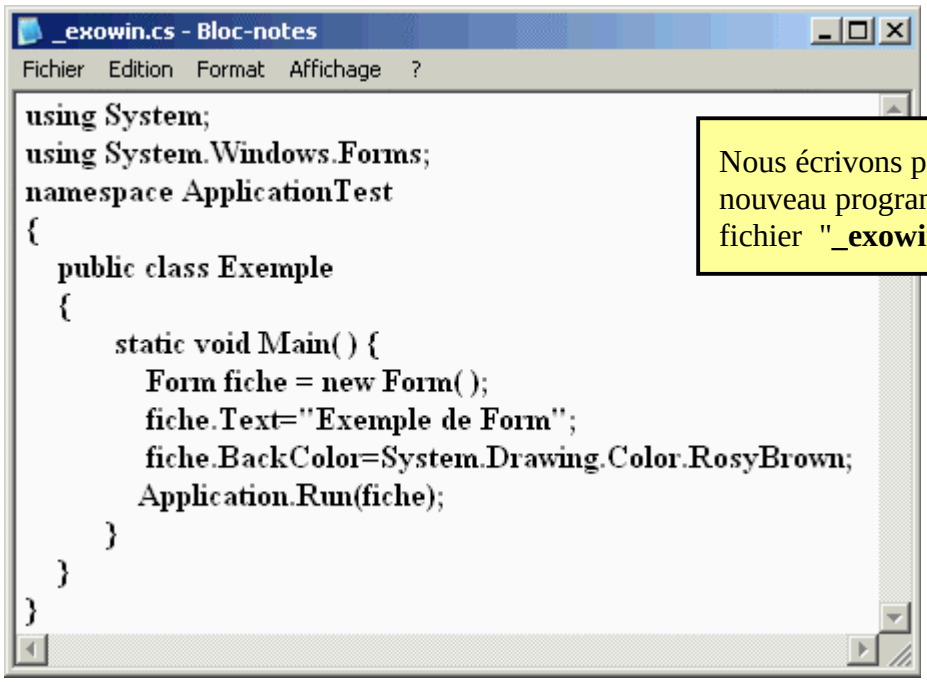
Nouveau fichier de commande **_exowin.bat** :

```
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) :

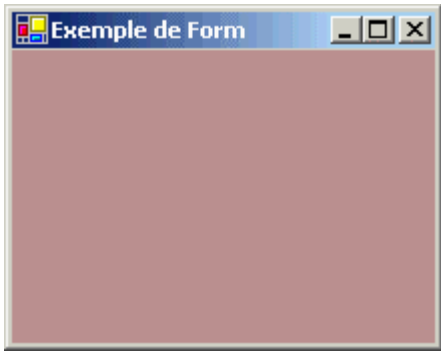


Cette fenêtre est un peu trop terne, nous pouvons travailler en mode console sur la fiche Form, afin par exemple de mettre un texte dans la barre de titre et de modifier sa couleur de fond.



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

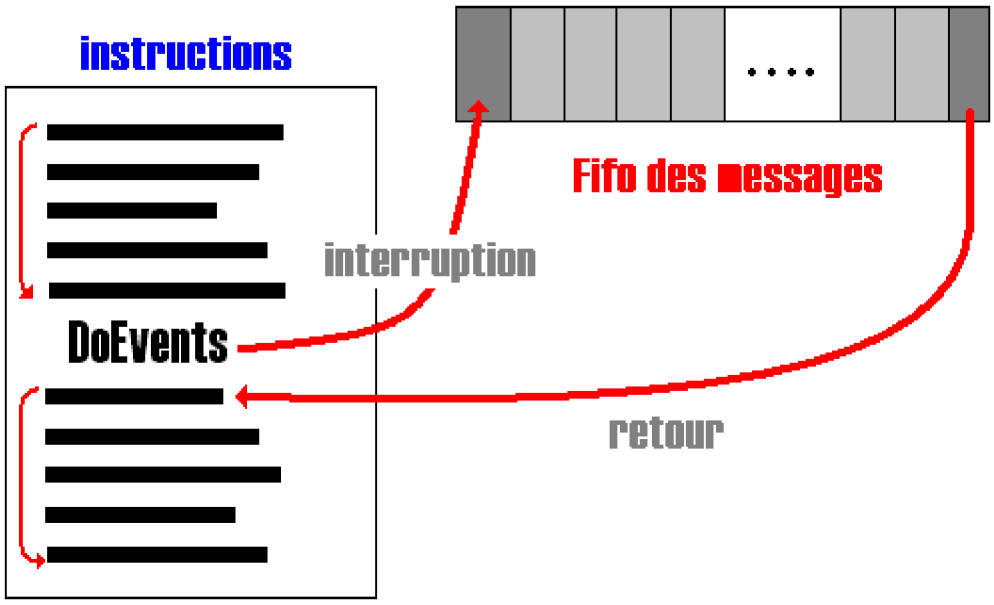
fiche.Show( );	il rend la fiche visible
}	fin de la méthode Main et donc tout est détruit et libéré et le processus est terminé.

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.

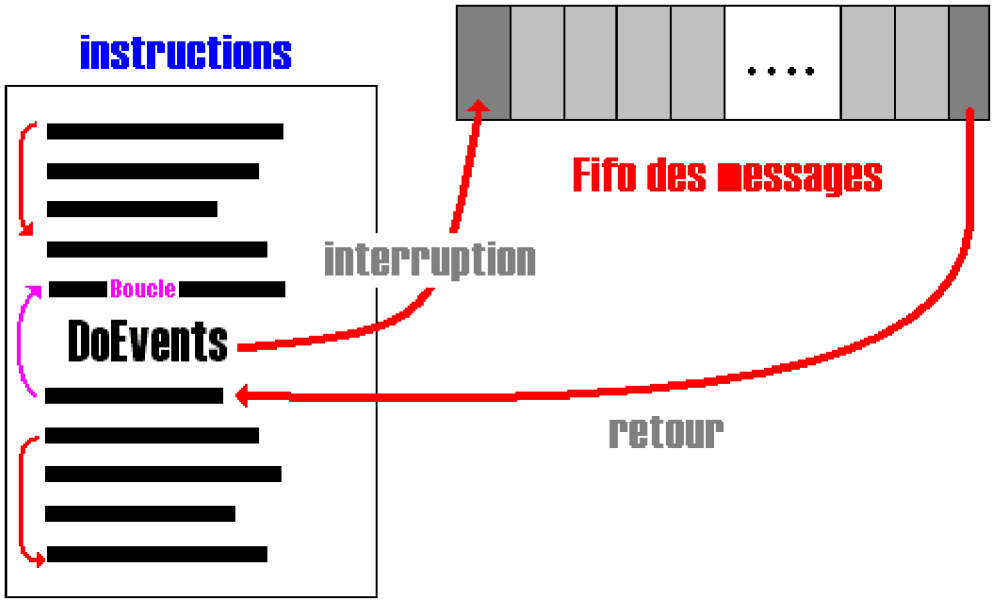
1.2.4 Que faire avec Application.DoEvents( )

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( );
```



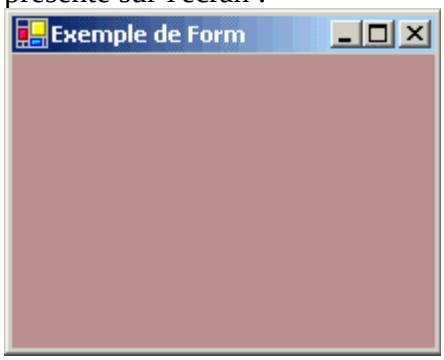
Il suffit que lorsque **DoEvents( )** s'exécute l'une des actions de traitement de messages provoque la mise du booléen stop à true pour que la boucle s'arrête et que le processus se termine.

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.

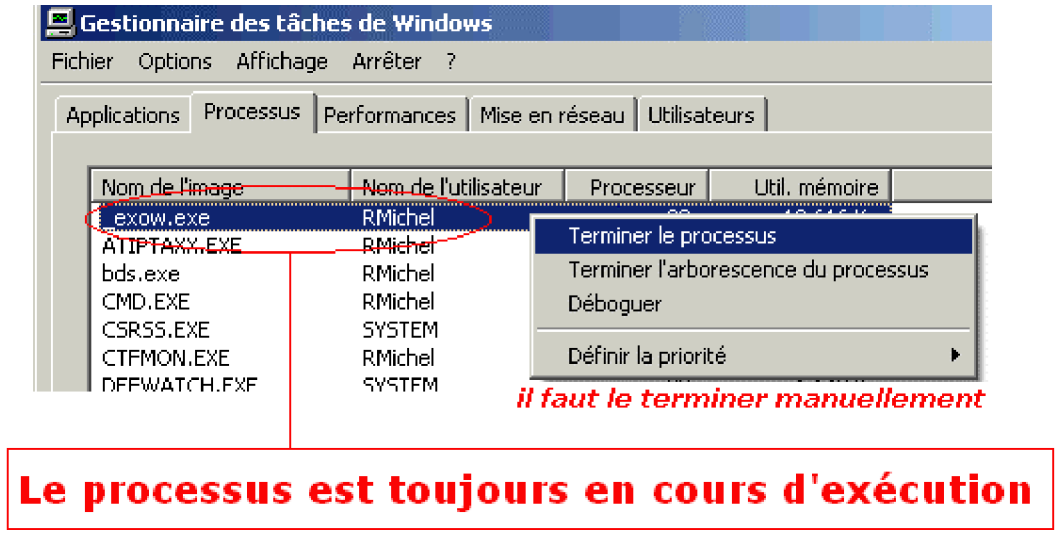
Attention à l'utilisation de la méthode Exit de la classe Application qui semblerait être utilisable dans ce cas, en effet cette méthode arrête toutes les boucles de messages en cours sur tous les threads et ferme toutes les fenêtres de l'application; mais cette méthode ne force pas la fermeture de l'application.

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

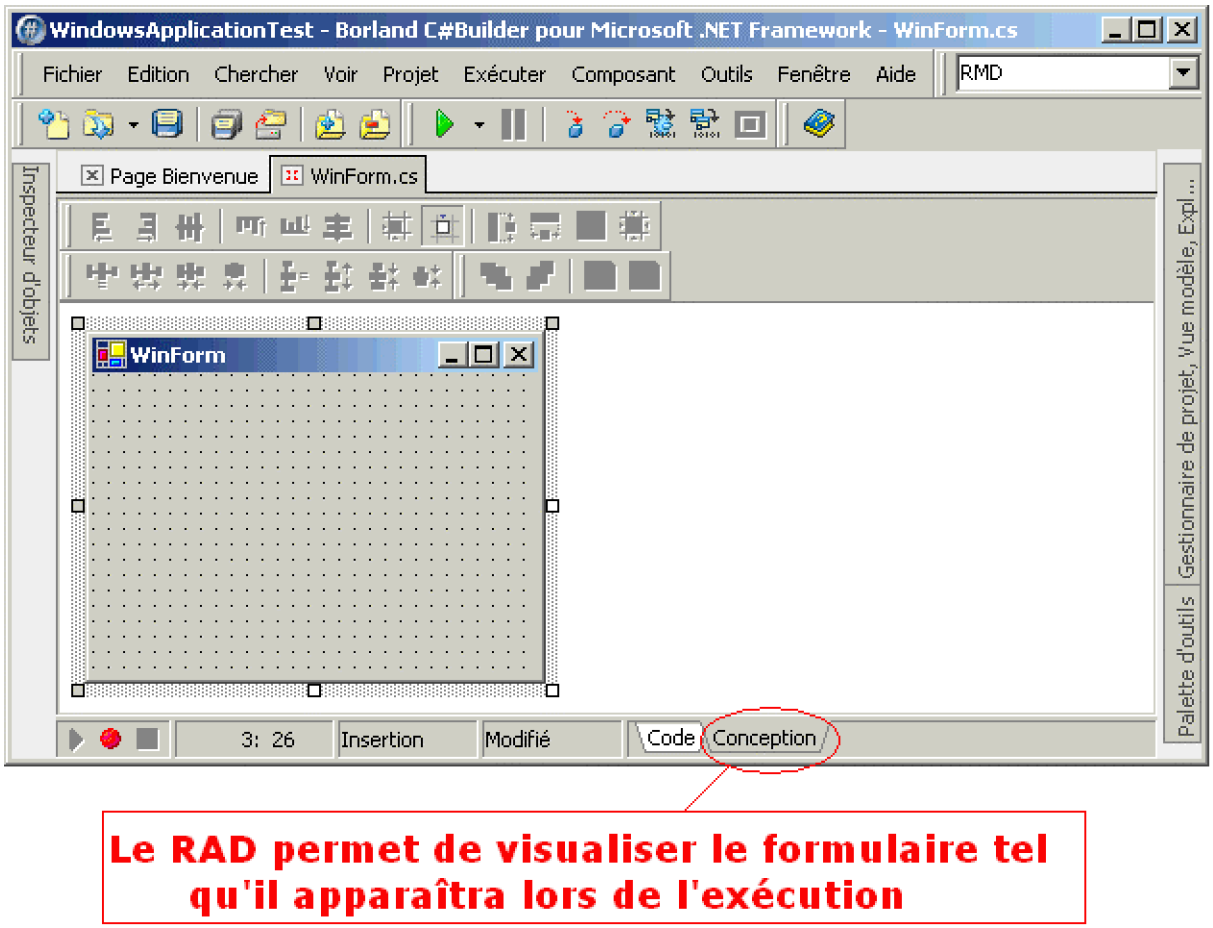
- **Visual Studio de microsoft** contient deux RAD de développement pour .Net, VBNet (fondé sur Visual Basic réellement objet et entièrement rénové) et Visual C# (fondé sur le langage C#), parfaitement adapté à .Net. *(prix très réduit pour l'éducation)*
- **C# Builder de Borland** reprenant les fonctionnalités de Visual C# dans Visual Studio, avec un intérêt supplémentaire pour un étudiant ou un apprenant : une version personnelle **gratuite** est téléchargeable.
- **Le monde de l'open source** construit un produit nommé **sharpDevelop** qui devrait à terme fournir à tous gratuitement aussi les mêmes fonctions.

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#.

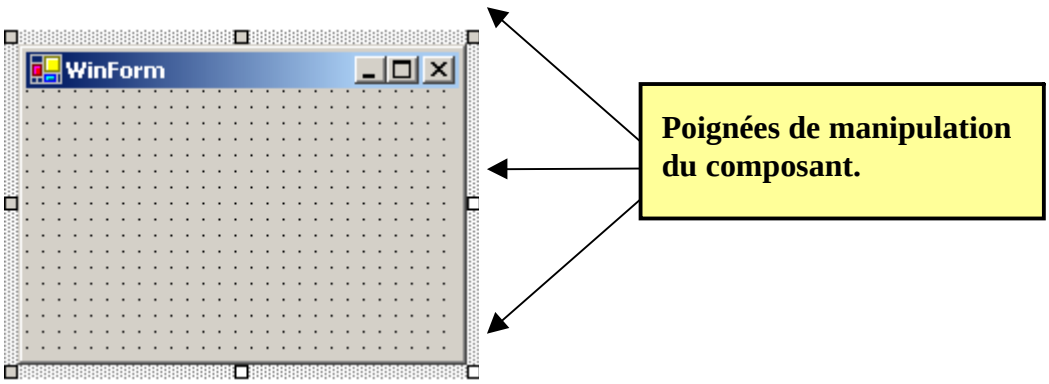
Etant donné la disponibilité gratuite du RAD C# Builder en version personnelle (très suffisante pour déjà écrire de bonnes applications) nous illustrerons tous nos exemples avec ce RAD.

Voici l'apparence d'un formulaire (ou fiche) dans le RAD C# Builder en mode conception :



Le RAD permet de visualiser le formulaire tel qu'il apparaîtra lors de l'exécution

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 :

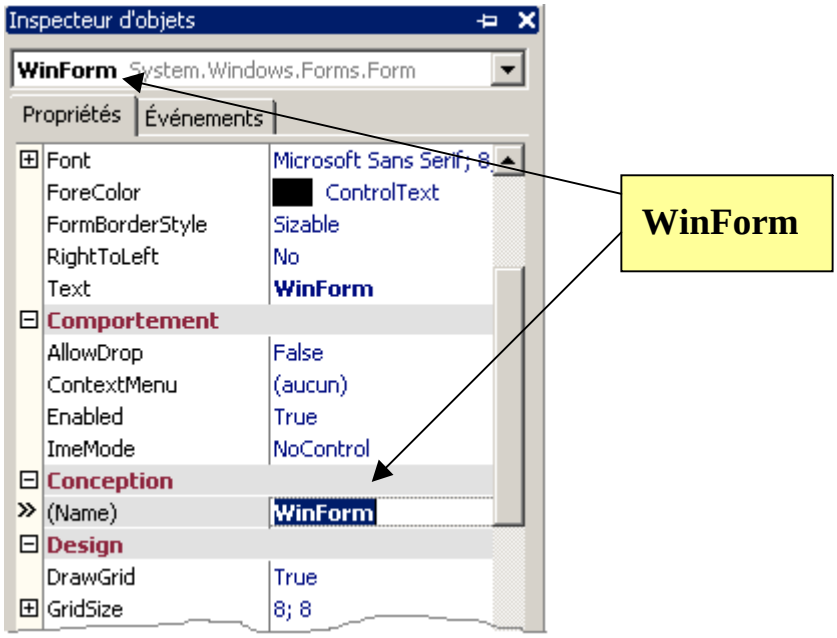
```
public Enum FormBorderStyle {Fixed3D, FixedDialog, FixedSingle, FixedToolWindow,
None, Sizable, SizableToolWindow  }
```

Dans un formulaire, le style est spécifié par la **propriété** FormBorderStyle de la classe Form :

```
public FormBorderStyle FormBorderStyle {get; set;}
```

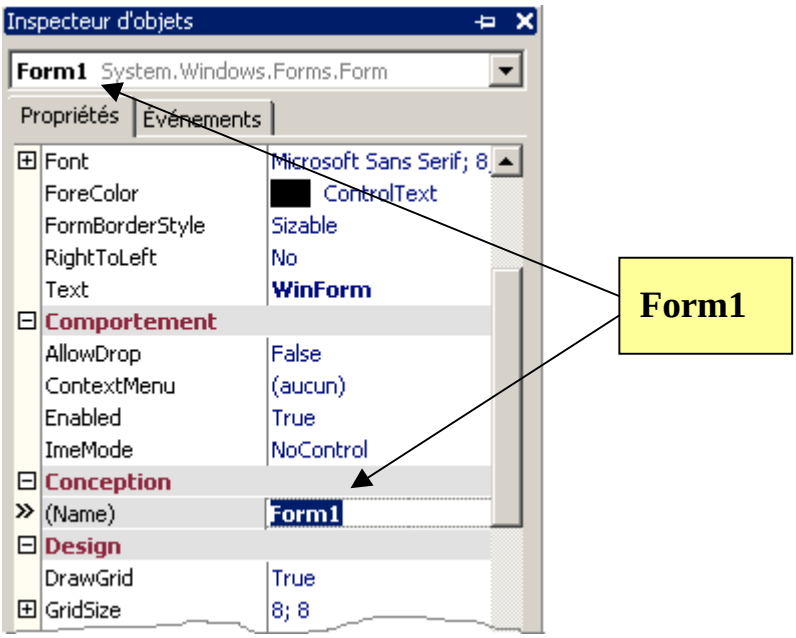
Toutes les propriétés en lecture et écriture d'une fiche sont accessibles à travers l'inspecteur d'objet qui répercute immédiatement en mode conception toute modification. Certaines provoquent des changements visuels d'autres non :

1°) changeons le nom d'identificateur de notre fiche dans le programme en modifiant la propriété Name qui vaut par défaut **WinForm** :

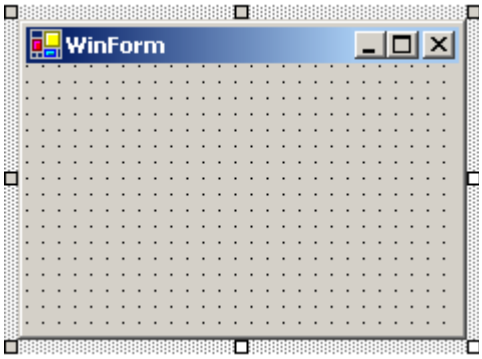


Le RAD construit automatiquement notre fiche principale comme une classe héritée de la classe Form et l'appelle WinForm :
public class WinForm : System.Windows.Forms.Form
{ ... }

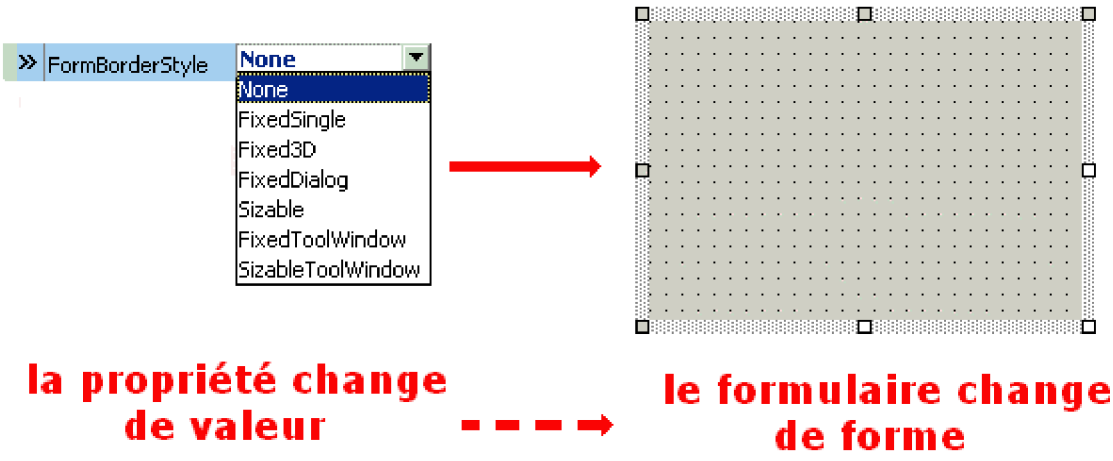
Après modification de la propriété Name par le texte **Form1**, nous obtenons :



La classe de notre formulaire s'appelle désormais **Form1**, mais son aspect visuel est resté le même :



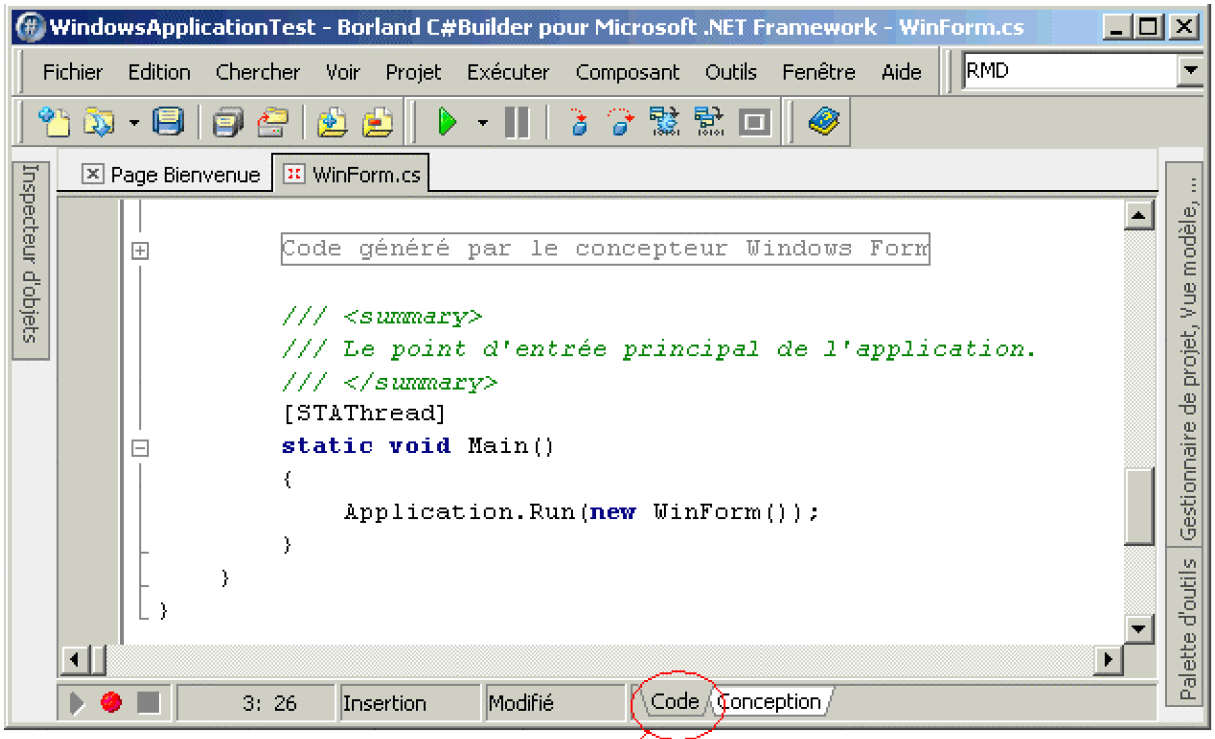
2°) Par exemple sélectionnons dans l'inspecteur d'objet de C# Builder, la propriété FormBorderStyle *(le style par défaut est* FormBorderStyle.Sizable *)* modifions la à la valeur **None** et regardons dans l'onglet conception la nouvelle forme de la fiche :



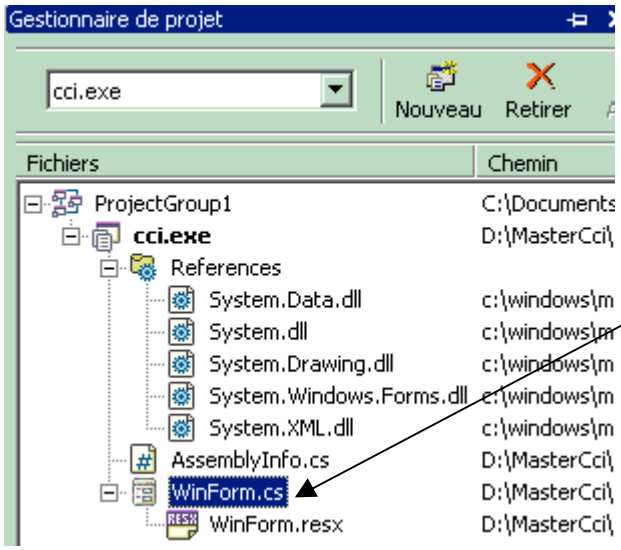
L'aspect visuel du formulaire a changé.

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 sa valeur initiale WinForm, voyons maintenant en supposant avoir appelé notre application **ProjApplication0** ce que C# Builder a engendré comme code source que nous trouvons dans l'onglet code pour notre formulaire :



Le RAD permet de visualiser le code source C# du formulaire



L'intégralité du code proposé par C# Builder est sauvegardé dans un fichier nommé WinForm.cs

Fichier WinForm.cs

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

namespace ProjApplication0
{

   public class WinForm : System.Windows.Forms.Form
   {

   }

}
```

Allure général du contenu de ce fichier affiché dans l'onglet code.

Fichier WinForm.cs

```
namespace ProjApplication0
{
    public class WinForm  : System.Windows.Forms.Form
    {
       public WinForm ()
       {
          ...
       }

       protected override void Dispose (bool disposing)
       { ...
       }

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

    }

}
```

La classe WinForm contient en première analyse 3 méthodes.

Premier éléments explicatifs de l'analyse du code :

La méthode	correspond
public WinForm ( ) **{** ... **}**	au constructeur d'objet de la classe WinForm et que la méthode
static void Main( ) **{** Application.Run (**new** WinForm ( )); **}**	au 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** WinForm ( )"de classe WinForm passé en paramètre à la méthode Run : c'est la fiche principale de l'application.
Application.Run (**new** WinForm ( )); 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 **WinForm** situé dans l'onglet code on se rend compte qu'il existe une ligne en grisé entre la méthode **Dispose** et la méthode **main** :

```
region Code généré par le concepteur Windows Form
```

Il s'agit en fait de code replié (masqué).

Voici le contenu exact de l'onglet code avec sa zone de code replié :

```
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 class WinForm : System.Windows.Forms.Form
    {
        /// <summary>
        /// Variable requise par le concepteur.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public WinForm ( )
        {
            //
            // Requis pour la gestion du concepteur Windows Form
            //
            InitializeComponent( );
            //
            // TODO: Ajouter tout le code du constructeur après l'appel de InitializeComponent
            //
```

```
        }
        /// <summary>
        /// Nettoyage des ressources utilisées.
        /// </summary>
        protected override void Dispose (bool disposing)
        {
            if (disposing)  {
                if (components != null) {
                    components.Dispose( );
                }
            }
            base.Dispose(disposing);
        }
        region Code généré par le concepteur Windows Form

        /// <summary>
        /// Le point d'entrée principal de l'application.
        /// </summary>
        [STAThread]
        static void Main( )
        {
            Application.Run(new WinForm ( ));
        }
    }
}
```

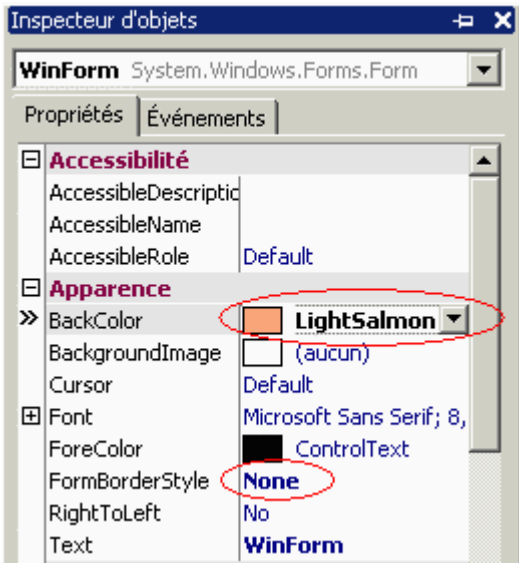
Si nous le déplions et nous voyons apparaître la méthode privée InitializeComponent( ) contenant du code qui a manifestement été généré directement. En outre cette méthode est appelée dans le constructeur d'objet WinForm :

```
public class WinForm  : System.Windows.Forms.Form
{
    public WinForm ()
    {
        InitializeComponent();
    } ••• votre code

    protected override void Dispose (bool disposing)
    { •••
    }

 # region Code généré par le concepteur Windows Form
    private void InitializeComponent()
    {
      code généré
      automatiquement
    }
 # region Code

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

Nous déplions la région Code généré par le concepteur Windows Form :

```
#region Code généré par le concepteur Windows Form
      /// <summary>
      /// Méthode requise pour la gestion du concepteur - ne pas modifier
      /// le contenu de cette méthode avec l'éditeur de code.
      /// </summary>
      private void InitializeComponent( )
      {
         //
         // WinForm
         //
         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
         this.ClientSize = new System.Drawing.Size(232, 157);
         this.Name = "WinForm";
         this.Text = "WinForm";
      }
#endregion
```

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 Code généré par le concepteur Windows Form
   /// <summary>
   /// Méthode requise pour la gestion du concepteur - ne pas modifier
   /// le contenu de cette méthode avec l'éditeur de code.
   /// </summary>
   private void InitializeComponent( )
   {
      //
      // WinForm
      //
```

```
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.BackColor = System.Drawing.Color.LightSalmon;
            this.ClientSize = new System.Drawing.Size(232, 157);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.None;
            this.Name = "WinForm";
            this.Text = "WinForm";
        }
        #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)  {
            if (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)  { // Nettoyage des ressources managées
            if (components != null) {
                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 class WinForm : System.Windows.Forms.Form
   {
      /// <summary>
       /// Variable requise par le concepteur.
       /// </summary>
      private System.ComponentModel.Container components = null;
      private UnComposant MonComposant = new UnComposant( );

      public WinForm ( )
      {

         InitializeComponent( );

         if ( components == null )
            components = new Container( );
         components.Add ( MonComposant );
      }
      /// <summary>
       /// Nettoyage des ressources utilisées.
       /// </summary>
      protected override void Dispose (bool disposing)
      {
         if (disposing)  {
            if (components != null) {
               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

      /// <summary>
      /// Le point d'entrée principal de l'application.
      /// </summary>
      [STAThread]
      static void Main( )
      {
         Application.Run(new WinForm ( ));
      }
   }
}
```

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 WinForm ( ));
}
```

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 WinForm ( ));
}
```

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 !

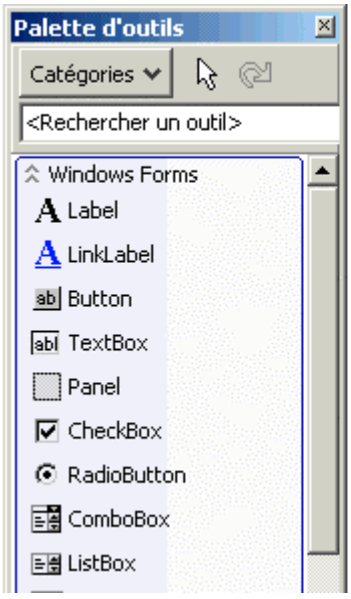
1.1 les composants en général

System.Windows.Forms.Control définit la classe de base des contrôles qui sont des composants avec représentation visuelle, les fiches en sont un exemple.

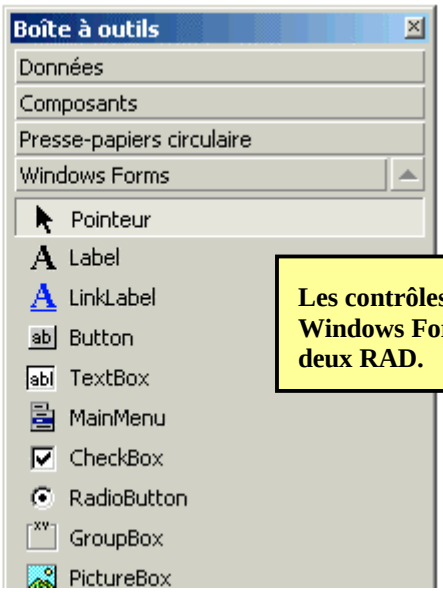
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ContainerControl
            System.Windows.Forms.Form

Dans les deux RAD Visual C# et C#Builder 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.

C# Builder :



Visual C# :



Les contrôles visuels ou Windows Forms dans les deux RAD.

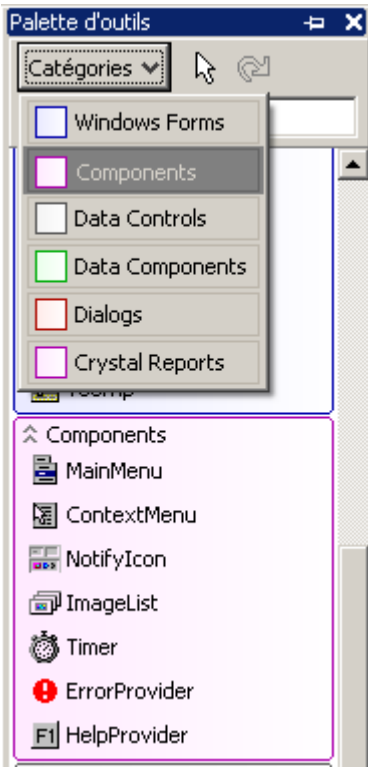
Cette distinction entre visuel et non visuel n'est pas très précise et dépend de la présentation dans le RAD. Cette variabilité de la dénomination n'a aucune importance pour l'utilisateur car tous les composants ont un fonctionnement du code identique dans les deux RAD.

En effet si nous prenons la classe System.Windows.Forms.MainMenu :

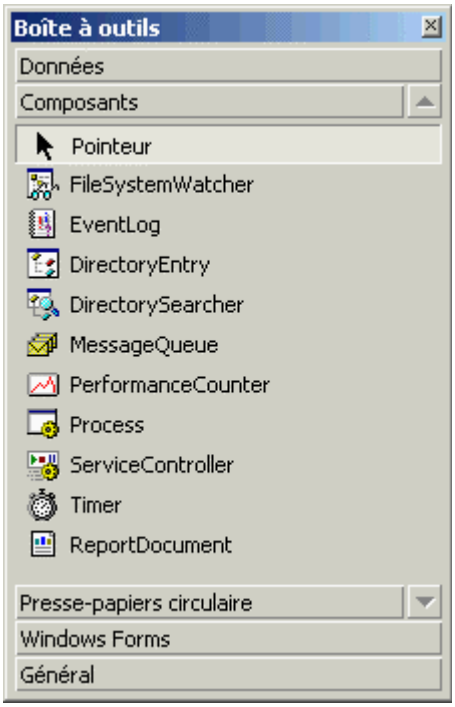
System.ComponentModel.Component
    System.Windows.Forms.Menu
      System.Windows.Forms.MainMenu

Nous voyons que Visual C# range System.Windows.Forms.MainMenu dans les contrôles alors que C# Builder le range dans les composants (donc non visuels). Ci-dessous les outils de composants proposés par les deux RAD :

C# Builder



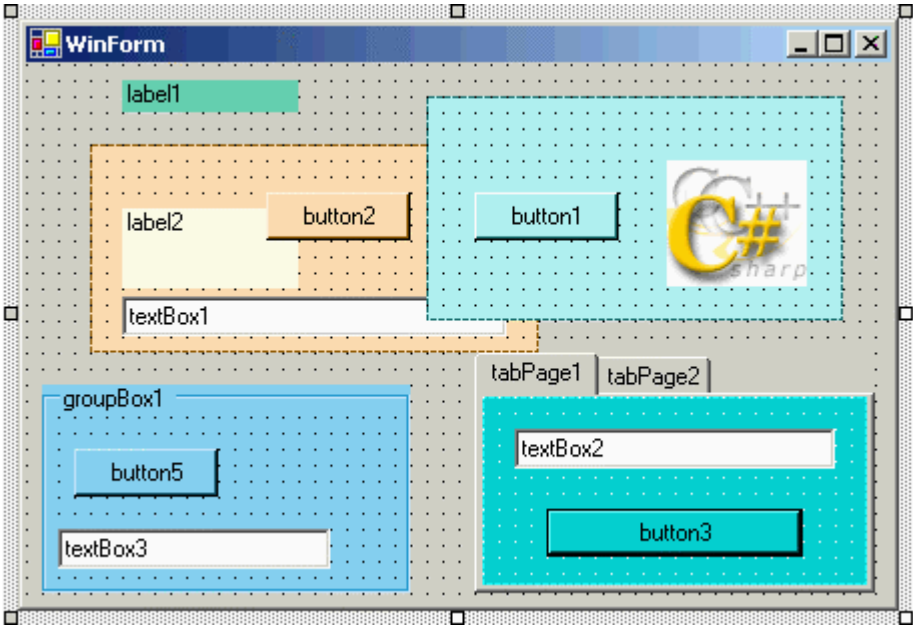
Visual C# :



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

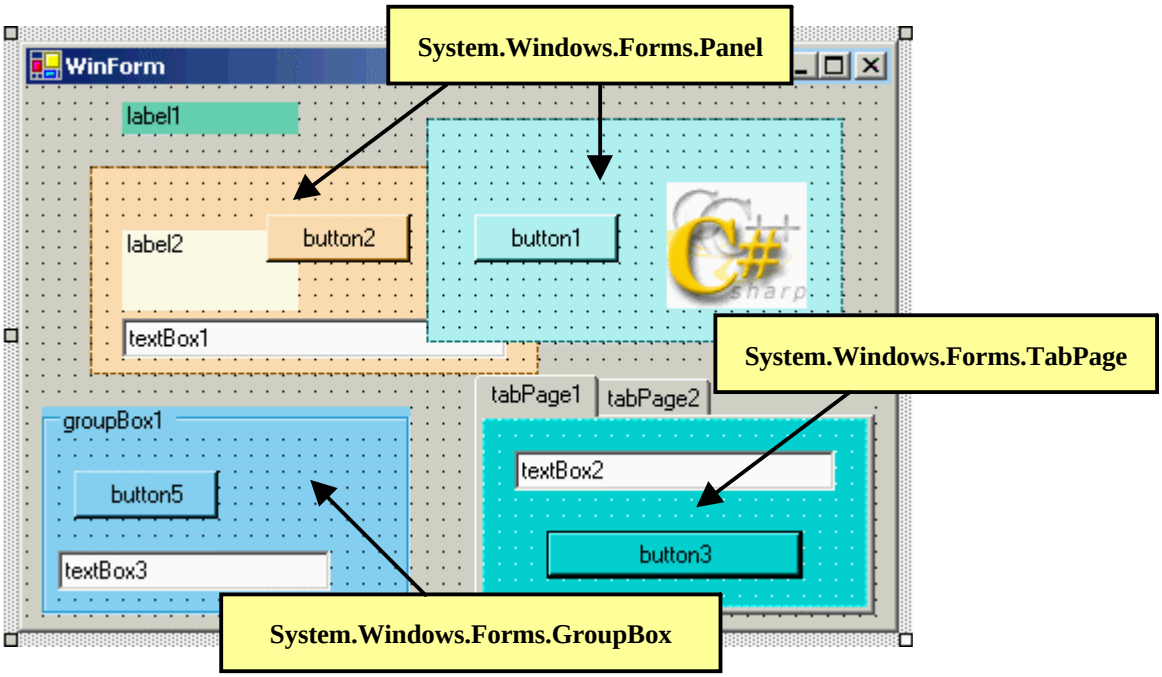
Voici un exemple de fiche comportant 7 catégories de contrôles différents :



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

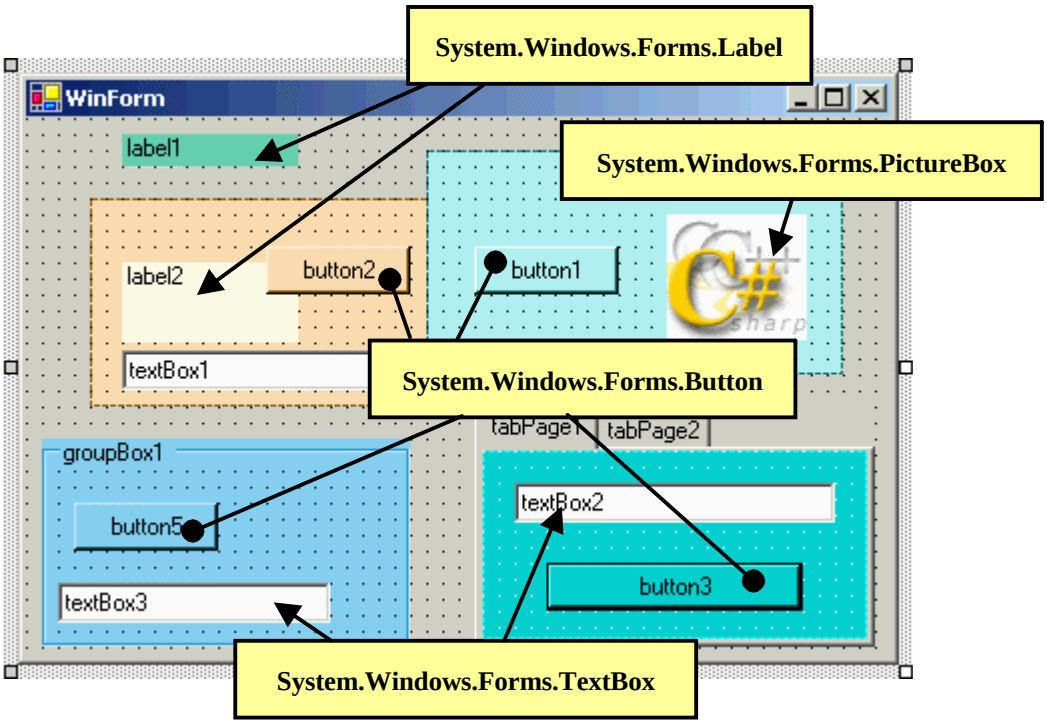
Sur la fiche précédente nous relevons outre le formulaire lui-même, quatre conteneurs visuels répartis en trois catégories de conteneurs visuels :



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 :



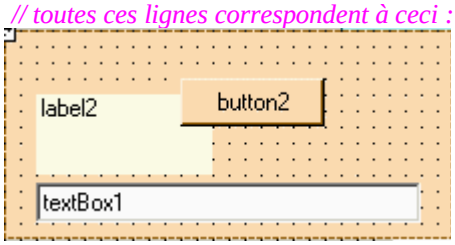
Le code C# engendré pour cette interface :

```
public class  WinForm  :  System .Windows.Forms.Form
{
  /// <summary>
 /// Variable requise par le concepteur.
 /// </summary>
 private  System .ComponentModel.Container components  =  null;
 private  System .Windows.Forms.Panel panel1 ;
 private  System .Windows.Forms.Label label1 ;
 private  System .Windows.Forms.Label label2 ;
 private  System .Windows.Forms.Panel panel2 ;
 private  System .Windows.Forms.PictureBox pictureBox1 ;
 private  System .Windows.Forms.Button button1 ;
 private  System .Windows.Forms.TextBox textBox1 ;
 private  System .Windows.Forms.Button button2 ;
 private  System .Windows.Forms.GroupBox groupBox1 ;
 private  System .Windows.Forms.TabControl tabControl1 ;
 private  System .Windows.Forms.TabPage tabPage1 ;
```

```
 private  System .Windows.Forms.TabPage tabPage2 ;
 private  System .Windows.Forms.Button button3 ;
 private  System .Windows.Forms.TextBox textBox2 ;
 private  System .Windows.Forms.ListBox listBox1 ;
 private  System .Windows.Forms.Button button4 ;
 private  System .Windows.Forms.Button button5 ;
 private  System .Windows.Forms.TextBox textBox3 ;
 private  System .Windows.Forms.MainMenu mainMenu1 ;

 public  WinForm ( ) {
   //
  // Requis pour la gestion du concepteur Windows Form
  //
   InitializeComponent ( );

   //
  // TODO: Ajouter tout le code du constructeur après l'appel de InitializeComponent
  //
  }

  /// <summary>
 /// Nettoyage des ressources utilisées.
 /// </summary>
  protected override void  Dispose  ( bool  disposing ) {
  if ( disposing )  {
   if ( components  !=  null)   {
     components.Dispose ( );
   }
  }
   base .Dispose ( disposing );
 }

  #region Code généré par le concepteur Windows Form
private void  InitializeComponent ( )
{
System .Resources.ResourceManager resources  =
        new  System .Resources.ResourceManager ( typeof ( WinForm ));
  this.panel1  =  new  System .Windows.Forms.Panel ( );
  this.button2  =  new  System .Windows.Forms.Button ( );
  this.textBox1  =  new  System .Windows.Forms.TextBox ( );
  this.label2  =  new  System .Windows.Forms.Label ( );
  this.label1  =  new  System .Windows.Forms.Label ( );
  this.panel2  =  new  System .Windows.Forms.Panel ( );
  this.button1  =  new  System .Windows.Forms.Button ( );
  this.pictureBox1  =  new  System .Windows.Forms.PictureBox ( );
  this.groupBox1  =  new  System .Windows.Forms.GroupBox ( );
  this.textBox3  =  new  System .Windows.Forms.TextBox ( );
  this.button5  =  new  System .Windows.Forms.Button ( );
  this.tabControl1  =  new  System .Windows.Forms.TabControl ( );
  this.tabPage1  =  new  System .Windows.Forms.TabPage ( );
  this.textBox2  =  new  System .Windows.Forms.TextBox ( );
  this.button3  =  new  System .Windows.Forms.Button ( );
  this.tabPage2  =  new  System .Windows.Forms.TabPage ( );
  this.button4  =  new  System .Windows.Forms.Button ( );
  this.listBox1  =  new  System .Windows.Forms.ListBox ( );
  this.mainMenu1  =  new  System .Windows.Forms.MainMenu ( );
  this.panel1.SuspendLayout ( );
  this.panel2.SuspendLayout ( );
  this.groupBox1.SuspendLayout ( );
  this.tabControl1.SuspendLayout ( );
  this.tabPage1.SuspendLayout ( );
```

```
this.tabPage2.SuspendLayout ( );
this.SuspendLayout ( );
```

```
//
// panel1
//
this.panel1.BackColor  = System .Drawing.Color.NavajoWhite ;
this.panel1.Controls.Add (this .button2 );
this.panel1.Controls.Add (this .textBox1 );
this.panel1.Controls.Add (this .label2 );
this.panel1.Location  = new  System .Drawing.Point ( 32, 40 );
this.panel1.Name  =  "panel1";
this.panel1.Size  = new  System .Drawing.Size ( 224, 104 );
this.panel1.TabIndex  =  0 ;
//
// button2
//
this.button2.Location  = new  System .Drawing.Point ( 88, 24 );
this.button2.Name  =  "button2";
this.button2.Size  = new  System .Drawing.Size ( 72, 24 );
this.button2.TabIndex  =  4 ;
this.button2.Text  =  "button2";
//
// textBox1
//
this.textBox1.Location  = new  System .Drawing.Point ( 16, 76 );
this.textBox1.Name  =  "textBox1";
this.textBox1.Size  = new  System .Drawing.Size ( 192, 20 );
this.textBox1.TabIndex  =  3 ;
this.textBox1.Text  =  "textBox1";
//
// label2
//
this.label2.BackColor  = System .Drawing.SystemColors.Info ;
this.label2.Location  = new  System .Drawing.Point ( 16, 32 );
this.label2.Name  =  "label2";
this.label2.Size  = new  System .Drawing.Size ( 88, 40 );
this.label2.TabIndex  =  2 ;
this.label2.Text  =  "label2";

// toutes ces lignes correspondent à ceci :
```



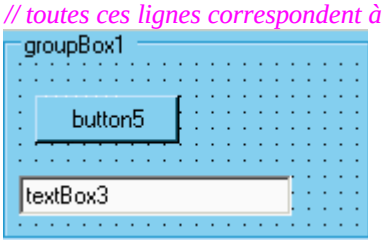

```
//
// panel2
//
this.panel2.BackColor  = System .Drawing.Color.PaleTurquoise ;
this.panel2.Controls.Add (this .button1 );
this.panel2.Controls.Add (this .pictureBox1 );
this.panel2.Location  = new  System .Drawing.Point ( 200, 16 );
this.panel2.Name  =  "panel2";
this.panel2.Size  = new  System .Drawing.Size ( 208, 112 );
this.panel2.TabIndex  =  2 ;
```

```
//
// button1
//
this.button1.Location = new System.Drawing.Point( 24, 48 );
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size( 72, 24 );
this.button1.TabIndex = 4;
this.button1.Text = "button1";
//
// pictureBox1
//
this.pictureBox1.BackColor = System.Drawing.Color.DarkKhaki;
this.pictureBox1.Image = (( System.Drawing.Image )( resources.GetObject
("pictureBox1.Image")));
// la ligne précédente charge l'image :
```

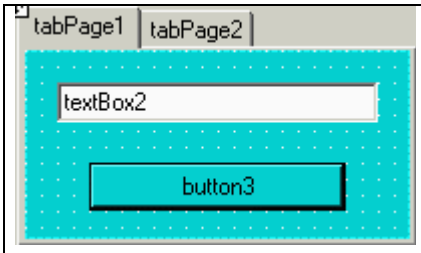
```
this.pictureBox1.Location = new System.Drawing.Point( 120, 32 );
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size( 70, 63 );
this.pictureBox1.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize;
this.pictureBox1.TabIndex = 0;
this.pictureBox1.TabStop = false;

// toutes ces lignes correspondent à ceci :
```


```
//
// groupBox1
//
this.groupBox1.BackColor = System.Drawing.Color.SkyBlue;
this.groupBox1.Controls.Add (this.textBox3 );
this.groupBox1.Controls.Add (this.button5 );
this.groupBox1.Location = new System.Drawing.Point( 8, 160 );
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size( 184, 104 );
this.groupBox1.TabIndex = 4;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "groupBox1";
//
// textBox3
//
this.textBox3.Location = new System.Drawing.Point( 8, 72 );
this.textBox3.Name = "textBox3";
this.textBox3.Size = new System.Drawing.Size( 136, 20 );
this.textBox3.TabIndex = 1;
this.textBox3.Text = "textBox3";
//
// button5
//
this.button5.Location = new System.Drawing.Point( 16, 32 );
```

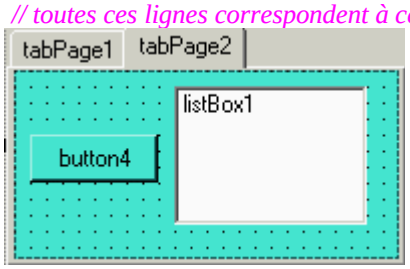
```
this.button5.Name = "button5";
this.button5.Size = new System.Drawing.Size( 72, 24 );
this.button5.TabIndex = 0;
this.button5.Text = "button5";

// toutes ces lignes correspondent à ceci :
```



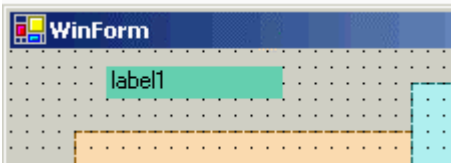

```
//
// tabControl1
//
this.tabControl1.Controls.Add(this.tabPage1);
this.tabControl1.Controls.Add(this.tabPage2);
this.tabControl1.Location = new System.Drawing.Point( 224, 144 );
this.tabControl1.Name = "tabControl1";
this.tabControl1.SelectedIndex = 0;
this.tabControl1.Size = new System.Drawing.Size( 200, 120 );
this.tabControl1.TabIndex = 5;
//
// tabPage1
//
this.tabPage1.BackColor = System.Drawing.Color.DarkTurquoise;
this.tabPage1.Controls.Add(this.textBox2);
this.tabPage1.Controls.Add(this.button3);
this.tabPage1.Location = new System.Drawing.Point( 4, 22 );
this.tabPage1.Name = "tabPage1";
this.tabPage1.Size = new System.Drawing.Size( 192, 94 );
this.tabPage1.TabIndex = 0;
this.tabPage1.Text = "tabPage1";
//
// textBox2
//
this.textBox2.Location = new System.Drawing.Point( 16, 16 );
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size( 160, 20 );
this.textBox2.TabIndex = 1;
this.textBox2.Text = "textBox2";
//
// button3
//
this.button3.Location = new System.Drawing.Point( 32, 56 );
this.button3.Name = "button3";
this.button3.Size = new System.Drawing.Size( 128, 24 );
this.button3.TabIndex = 0;
this.button3.Text = "button3";

// toutes ces lignes correspondent à ceci :
```



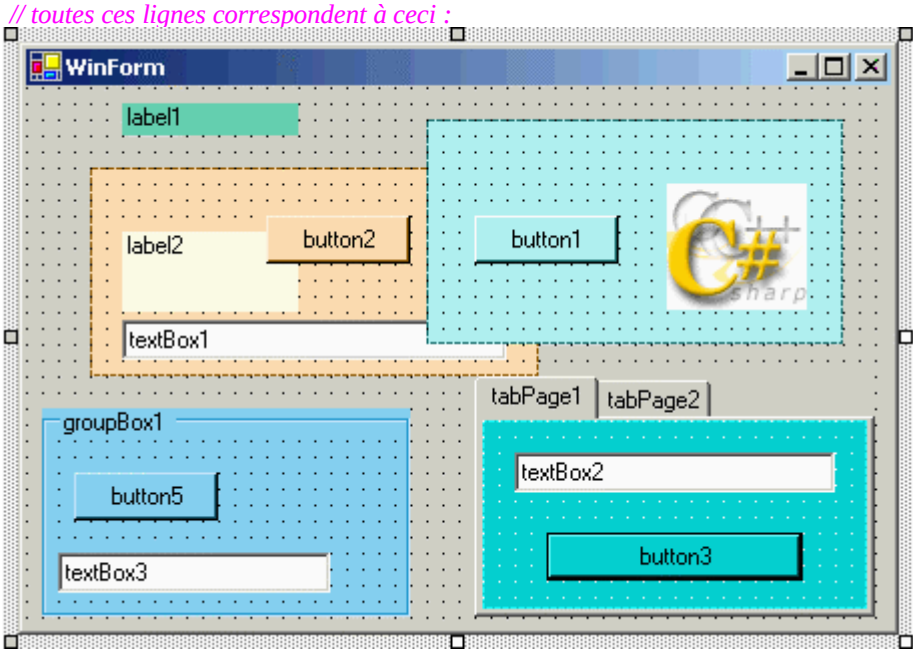

```
//
// tabPage2
//
this.tabPage2.BackColor = System.Drawing.Color.Turquoise;
this.tabPage2.Controls.Add(this.button4);
this.tabPage2.Controls.Add(this.listBox1);
this.tabPage2.Location = new System.Drawing.Point(4, 22);
this.tabPage2.Name = "tabPage2";
this.tabPage2.Size = new System.Drawing.Size(192, 94);
this.tabPage2.TabIndex = 1;
this.tabPage2.Text = "tabPage2";
//
// button4
//
this.button4.Location = new System.Drawing.Point(8, 32);
this.button4.Name = "button4";
this.button4.Size = new System.Drawing.Size(64, 24);
this.button4.TabIndex = 1;
this.button4.Text = "button4";
//
// listBox1
//
this.listBox1.Location = new System.Drawing.Point(80, 8);
this.listBox1.Name = "listBox1";
this.listBox1.Size = new System.Drawing.Size(96, 69);
this.listBox1.TabIndex = 0;

// toutes ces lignes correspondent à ceci :
```




```
//
// label1
//
this.label1.BackColor = System.Drawing.Color.MediumAquamarine;
this.label1.Location = new System.Drawing.Point(48, 8);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(88, 16);
this.label1.TabIndex = 1;
this.label1.Text = "label1";

// les 6 lignes précédentes correspondent à ceci :
```




```
//
// WinForm
//
this.AutoScaleBaseSize = new System.Drawing.Size( 5, 13 );
this.ClientSize = new System.Drawing.Size( 432, 269 );
this.Controls.Add (this.tabControl1 );
this.Controls.Add (this.groupBox1 );
this.Controls.Add (this.panel2 );
this.Controls.Add (this.label1 );
this.Controls.Add (this.panel1 );
this.Menu = this.mainMenu1 ;
this.Name = "WinForm";
this.Text = "WinForm";
this.panel1.ResumeLayout ( false );
this.panel2.ResumeLayout ( false );
this.groupBox1.ResumeLayout ( false );
this.tabControl1.ResumeLayout ( false );
this.tabPage1.ResumeLayout ( false );
this.tabPage2.ResumeLayout ( false );
this.ResumeLayout ( false );

// toutes ces lignes correspondent à ceci :
```




```
}
  #endregion
```

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 ){ }
```

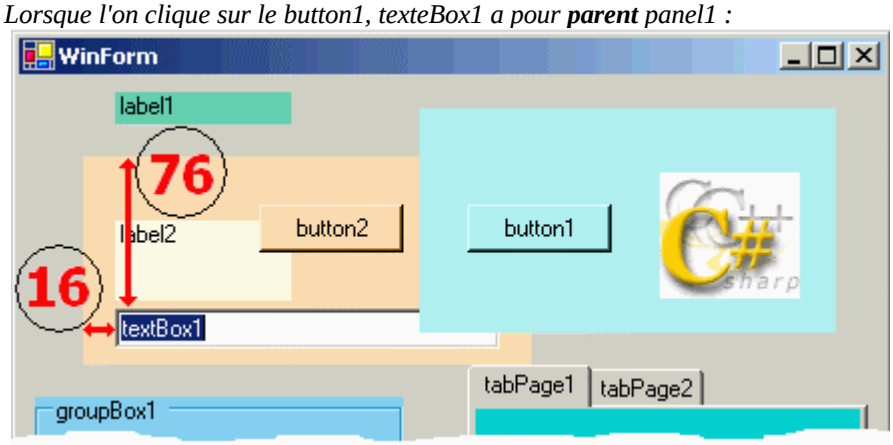
Les lignes d'abonnement sont engendrées dans la méthode InitializeComponent ( ) :

```
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)* :

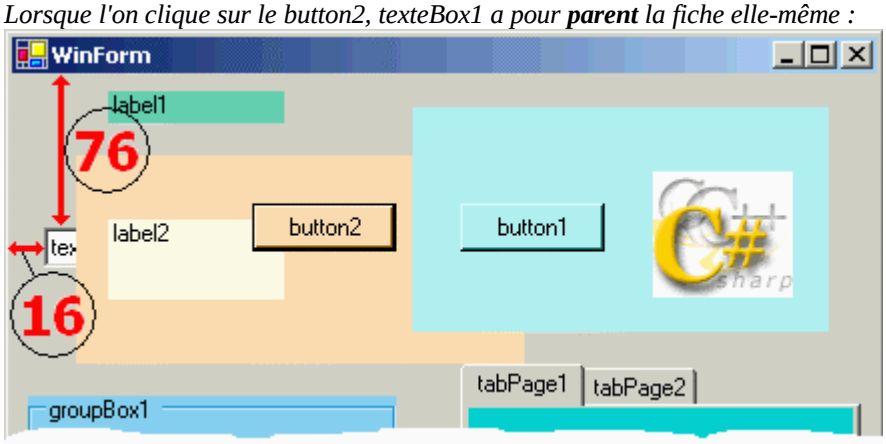
```
// Gestionnaire du click sur button1 :

private void button1_Click ( object sender, System .EventArgs e ){
  textBox1.Parent = panel1 ;
}
```

*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 :

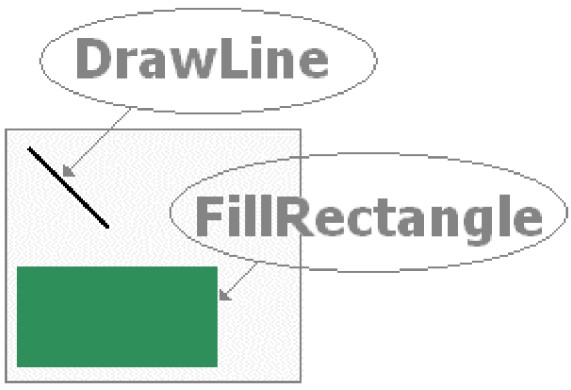
Bibliothèque de classes .NET Framework

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.

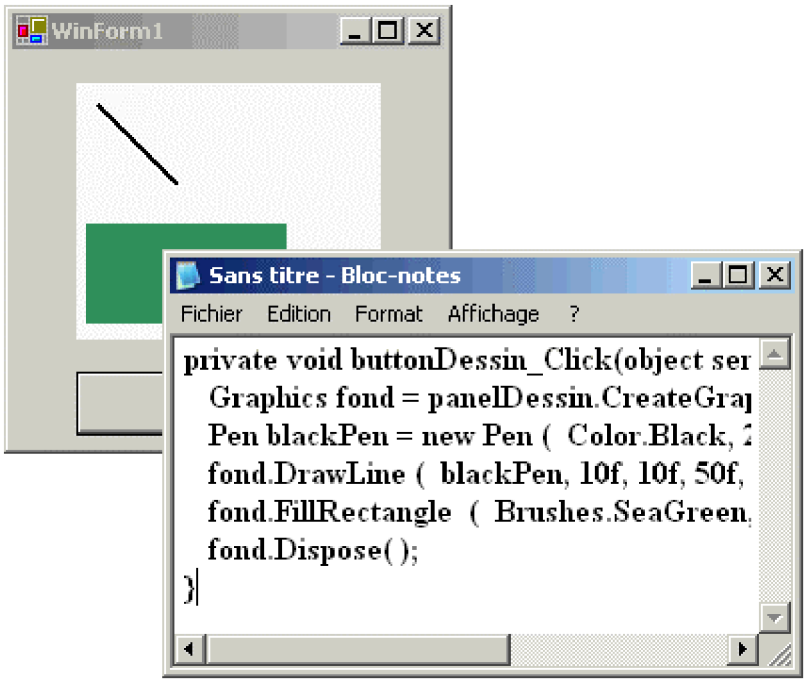
Nous supposons disposer d'un formulaire nommé WinForm1 contenant un panneau nommé panelDessin (**private System.Windows.Forms.Panel** panelDessin) et un bouton nommé buttonDessin (**private System.Windows.Forms.Button** buttonDessin)

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( );
}
```

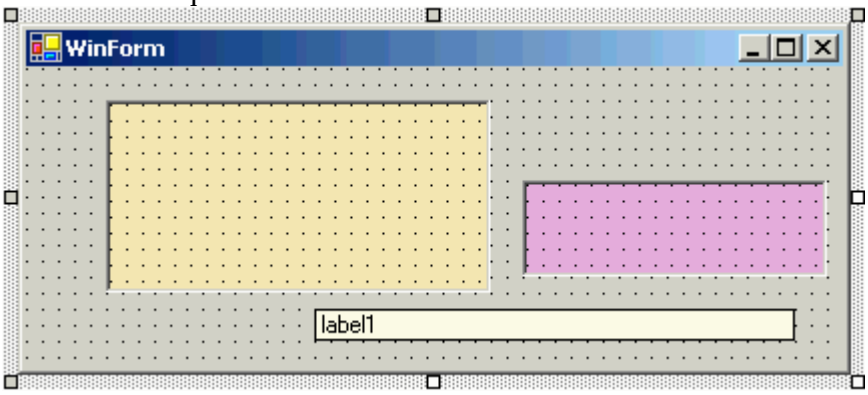
Le RAD a enregistré (abonné) le gestionnaire WinForm1_Paint auprès du délégué Paint dans le corps de laméthode InitializeComponent :

```
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 WinForm et deux panel (panel2 : jaune foncé et panel1 : violet), la label1 ne sert qu'à afficher du texte en sortie :

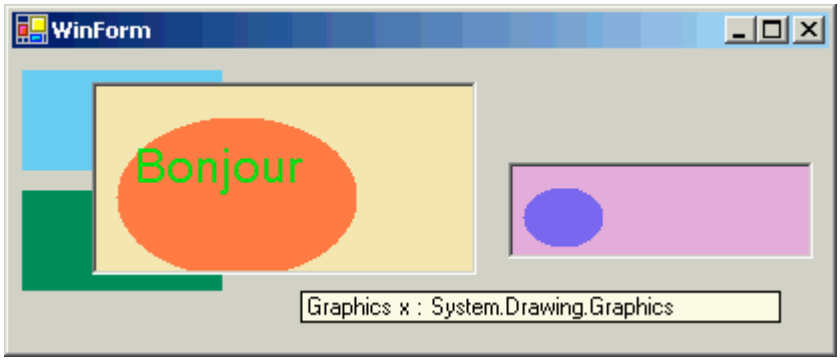


Nous écrivons une méthode **TracerDessin** permettant de dessiner sur le fond de la fiche, sur le fond des deux panel et d'écrire du texte sur le fond d'un panel. La méthode **TracerDessin** est appelée dans le gestionnaire de l'événement Paint du formulaire lors de son redessinement de la fiche afin d'assurer la persistance de tous les traçés. Voici le code que nous créons :

```
private void InitializeComponent( )  {
  ...
  this.Paint += new System.Windows.Forms.PaintEventHandler ( this.WinForm_Paint );
  ...
}

private void WinForm_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 );
```

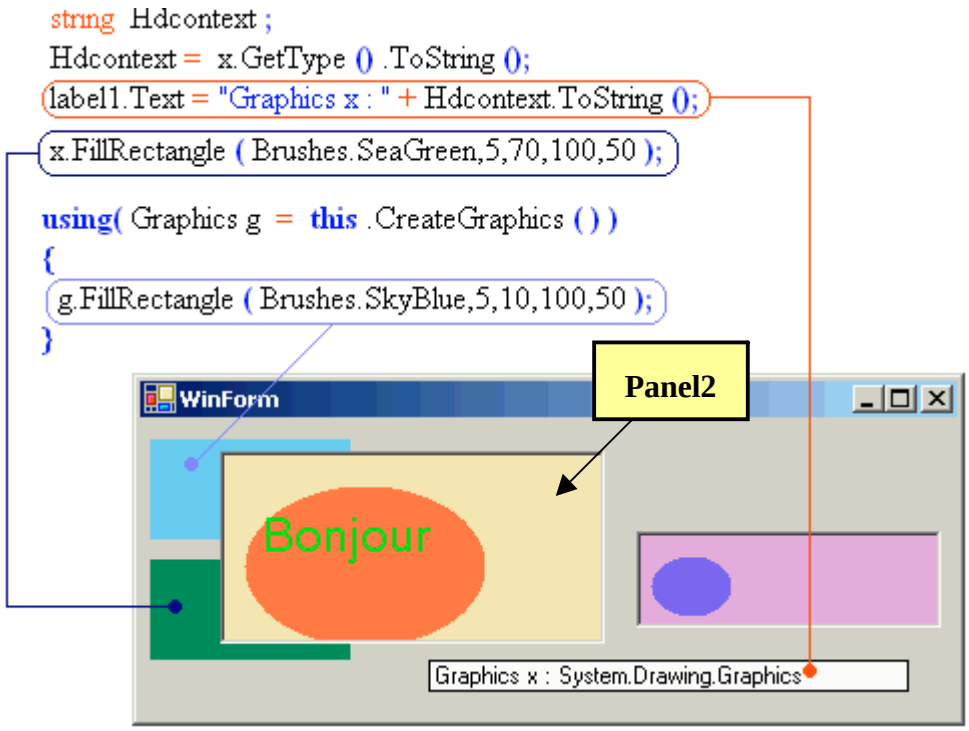
```
 }
}
```

L'exécution de code produit l'affichage suivant :



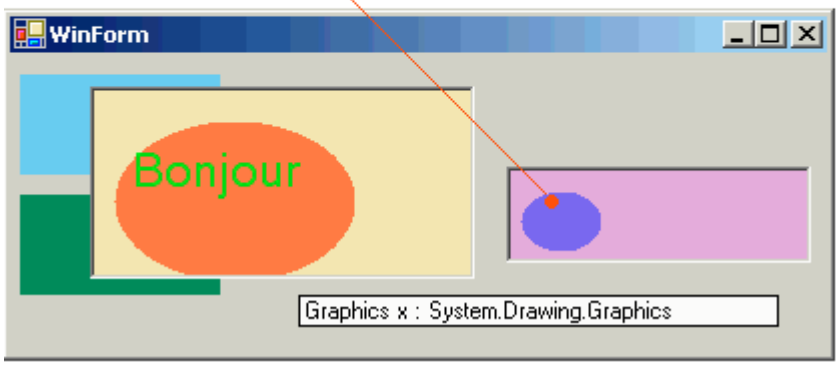
Illustrons les actions de dessin de chaque ligne de code de la méthode TracerDessin :

```
private void  TracerDessin ( Graphics x  ) {
```



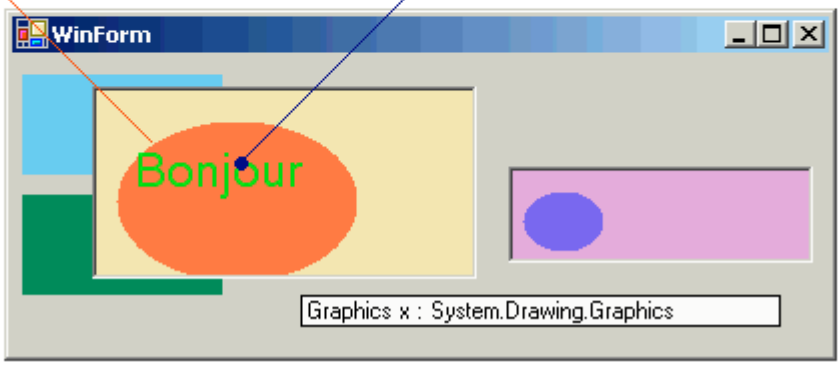
On dessine deux rectangles sur le fond de la fiche, nous notons que ces deux rectangles ont une intersection non vide avec le panel2 (jaune foncé)et que cela n'altère pas le dessin du panel. En effet le panel est un contrôle et donc se redessine lui-même. Nous en déduisons que le fond graphique est situé "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 droite sur le fond du panel1

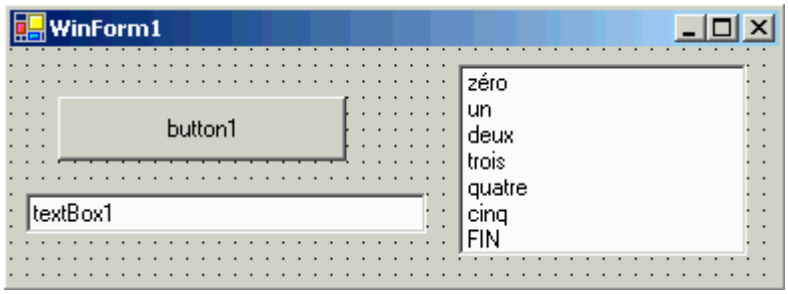
```
using( Graphics h = panel2.CreateGraphics () )
{
h.FillEllipse ( Brushes.Tomato,10,15,120,80 );
h.DrawString ("Bonjour" , new Font (this .Font.FontFamily.Name,18 ) ,Brushes.Lime,15,25 );
}
}
```



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 :



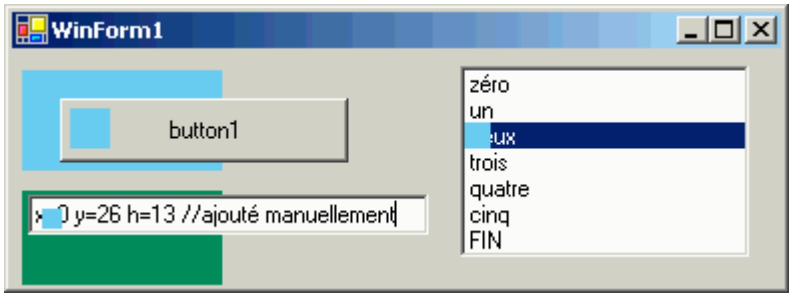
Les graphiques sont gérés dans le code ci-après :

```
namespace ProjWindowsApplication2
{
   /// <summary>
   /// Description Résumé de WinForm1.
   /// </summary>
   public class WinForm1 : System.Windows.Forms.Form
   {
      /// <summary>
      /// Variable requise par le concepteur.
      /// </summary>
      private System.ComponentModel.Container components = null ;
      private System.Windows.Forms.ListBox listBox1;
      private System.Windows.Forms.TextBox textBox1;
      private System.Windows.Forms.Button button1;

      private WinForm1( )
      {
         //
         // Requis pour la gestion du concepteur Windows Form
         //
         InitializeComponent();
         //
         // TODO: Ajouter tout le code du constructeur après l'appel de InitializeComponent
         //
      }
      /// <summary>
      /// Nettoyage des ressources utilisées.
      /// </summary>
      protected override void Dispose (bool disposing)
      {
         if (disposing)
         {
            if (components != null)
            {
               components.Dispose();
            }
         }
         base.Dispose(disposing);
      }

      #region Code généré par le concepteur Windows Form
      /// <summary>
      /// Méthode requise pour la gestion du concepteur - ne pas modifier
      /// le contenu de cette méthode avec l'éditeur de code.
      /// </summary>
      private void InitializeComponent( )
      {
         this.listBox1 = new System.Windows.Forms.ListBox( );
```

```
        this.textBox1 = new System.Windows.Forms.TextBox( );
        this.button1 = new System.Windows.Forms.Button( );
        this.SuspendLayout( );
        //
        // listBox1
        //
        this.listBox1.Items.AddRange ( new object[] {
                "zéro",
                "un",
                "deux",
                "trois",
                "quatre",
                "cinq",
                "FIN" }  );
        this.listBox1.Location = new System.Drawing.Point(224, 8);
        this.listBox1.Name = "listBox1";
        this.listBox1.Size = new System.Drawing.Size(144, 95);
        this.listBox1.TabIndex = 9;
        this.listBox1.SelectedIndexChanged +=
                                new System.EventHandler ( this.listBox1_SelectedIndexChanged );
        //
        // textBox1
        //
        this.textBox1.Location = new System.Drawing.Point(8, 72);
        this.textBox1.Name = "textBox1";
        this.textBox1.Size = new System.Drawing.Size(200, 20);
        this.textBox1.TabIndex = 8;
        this.textBox1.Text = "textBox1";
        this.textBox1.TextChanged += new System.EventHandler(this.textBox1_TextChanged);
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(24, 24);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(144, 32);
        this.button1.TabIndex = 7;
        this.button1.Text = "button1";
        this.button1.Paint += new System.Windows.Forms.PaintEventHandler(this.button1_Paint);
        //
        // WinForm1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(384, 117);
        this.Controls.Add(this.listBox1);
        this.Controls.Add(this.textBox1);
        this.Controls.Add(this.button1);
        this.Name = "WinForm1";
        this.StartPosition = System.Windows.Forms.FormStartPosition.Manual;
        this.Text = "WinForm1";
        this.Paint += new System.Windows.Forms.PaintEventHandler(this.WinForm1_Paint);
        this.ResumeLayout(false);
    }
    #endregion

    //-- dessin persistant sur le fond de la fiche par gestionnaire Paint:
    private void WinForm1_Paint(object sender, System.Windows.Forms.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, System.Windows.Forms.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, System.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, System.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 :

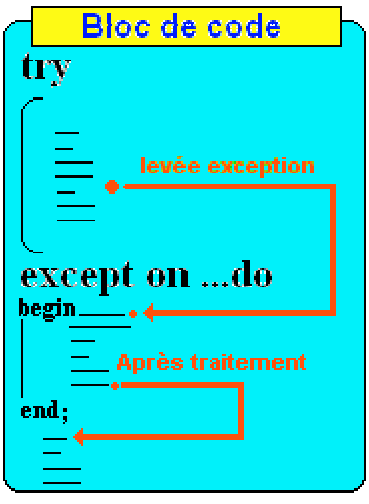
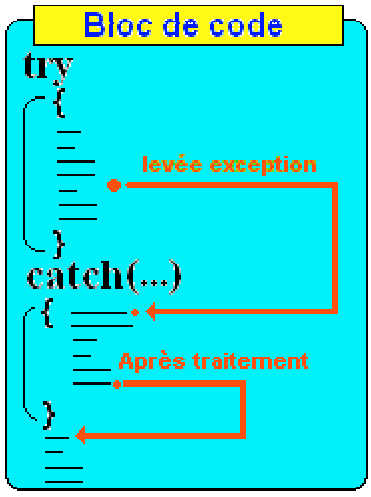
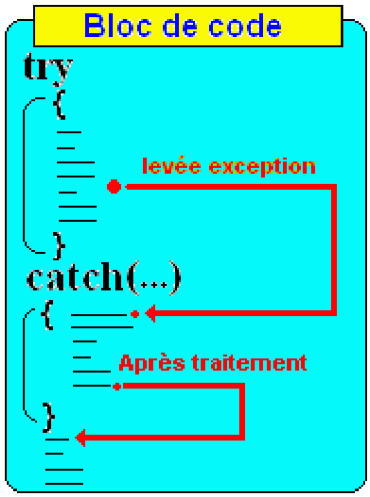


Exceptions : C# comparé à Java et Delphi

1. similitude et différence

Afin de ne pas alourdir l'ouvrage déjà volumineux, nous ne développons pas de cours spécial pour les exceptions en C# : 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.

C'est pourquoi nous renvoyons le lecteur au chapitre 9.3.2 (exceptions en java) 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 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> - ... **}** **fonctionnement identique à C#**  	**try** **{** - ... <lignes de code à protéger> - ...**}** **catch** (ExxException E) **{**- ... <lignes de code réagissant à l'exception> - ... **}** **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

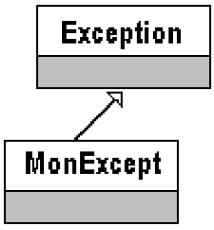
Dans les 3 langages la classes de base des exceptions se nomme : **Exception**

- 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#).

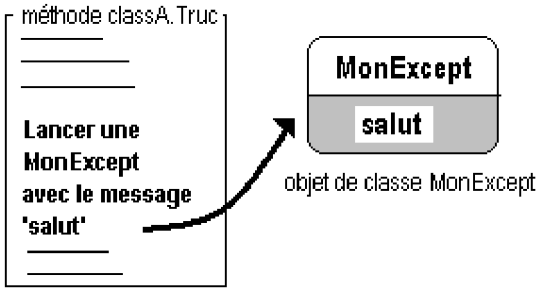
- Le mécanisme général d'interception des exceptions à travers des gestionnaires d'exceptions **try…except** ou **try…catch** s'applique à tous les types d'exceptions y compris les exceptions personnalisées.

1°) Création d'une nouvelle classe :



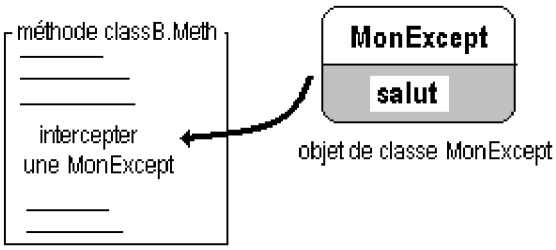
Delphi	Java	C#
Type MonExcept = **class** (Exception) .... **End;** MonExcept hérite par construction de la propriété **public** Message de sa mère, en lecture et écriture.	**class** MonExcept **extends** Exception **{** **public** MonExcept (String s) { **super**(s); ..... } **}**	**class** MonExcept : 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 ( ) {

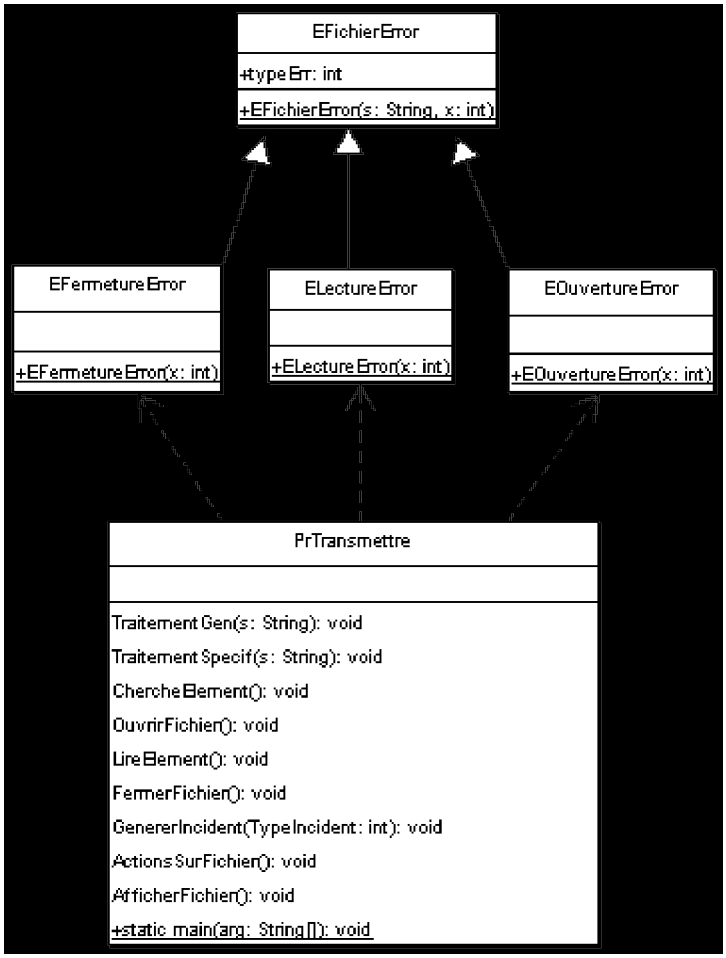
```
Procedure classB.Meth;
begin
  …..
   try
     ….. // code à protéger
   except on MonExcept do
     begin
      ….. // code de réaction à l'exception
     end;
  end;

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
     }
}
```

3. Un exemple de traitement en C#

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 :

<**programme**>

…<**ActionsSurFichier**>

…..<**AfficheFichier**>

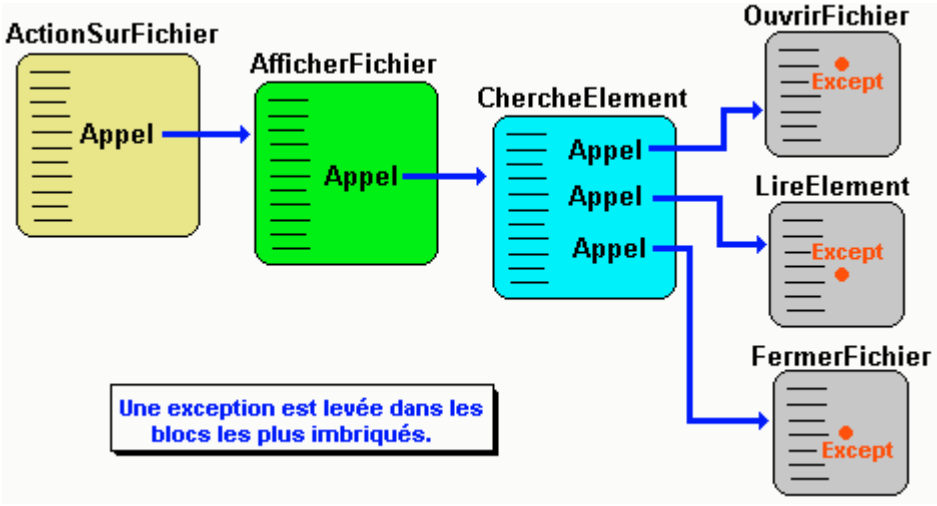
……..<**ChercheElement**>

.....<***OuvrirFichier***> **--> exception**

.....<***LireElement***> **--> exception**

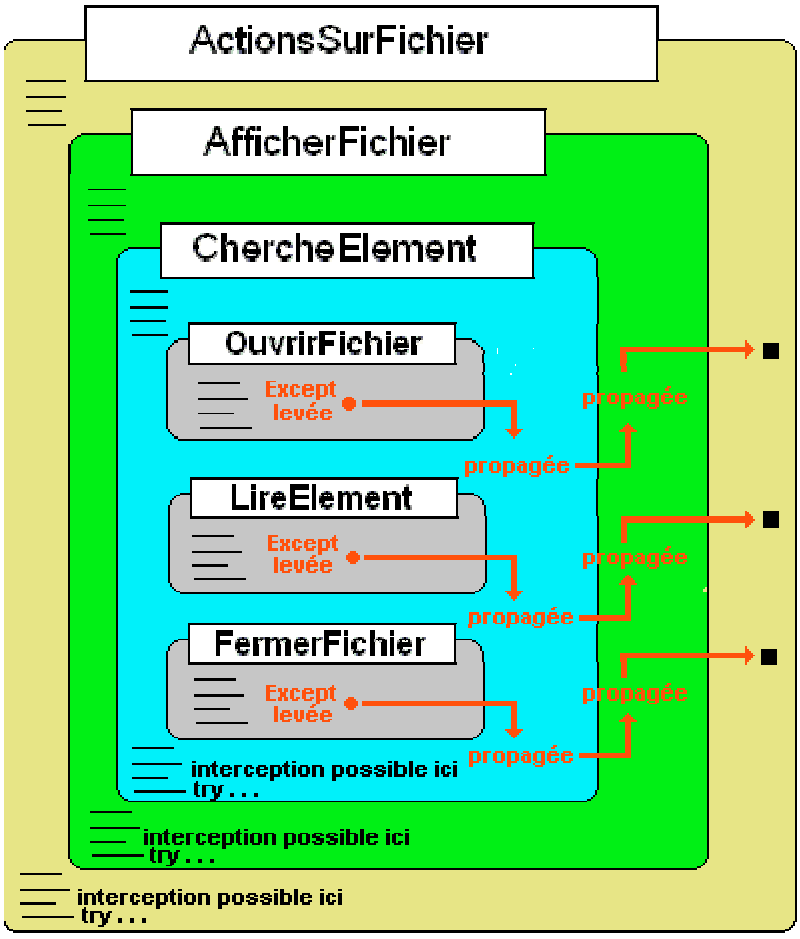
…….....<***FermerFichier***> **--> exception**

Les trois blocs du dernier niveau les plus internes <***OuvrirFichier***>, <***LireElement***> et <***FermerFichier***> peuvent lancer chacun une exception selon le schéma ci-après :



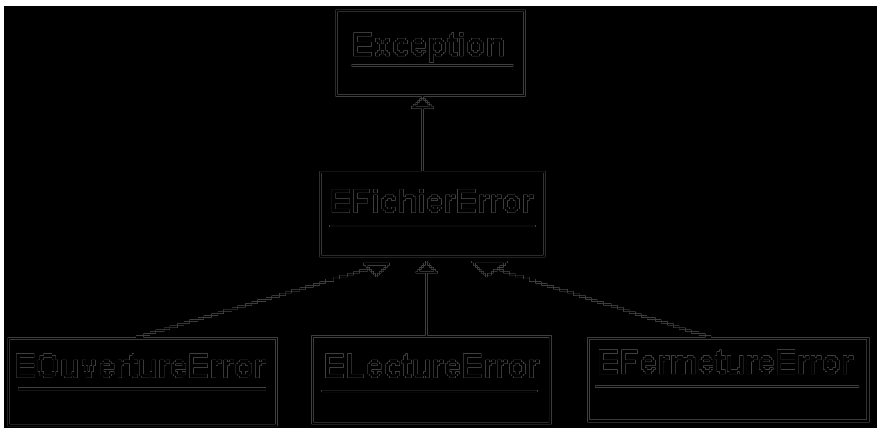
- ***La démarche***

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.



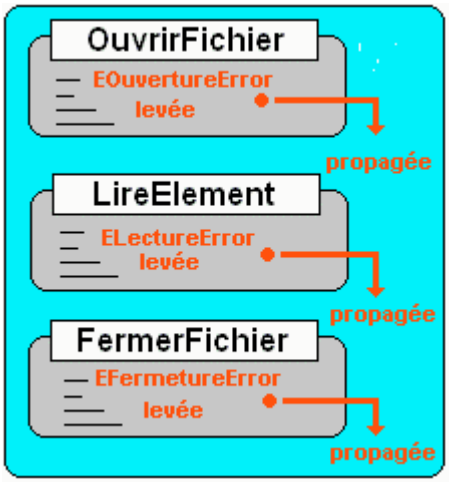
- ***Les classes d'exception***

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 :



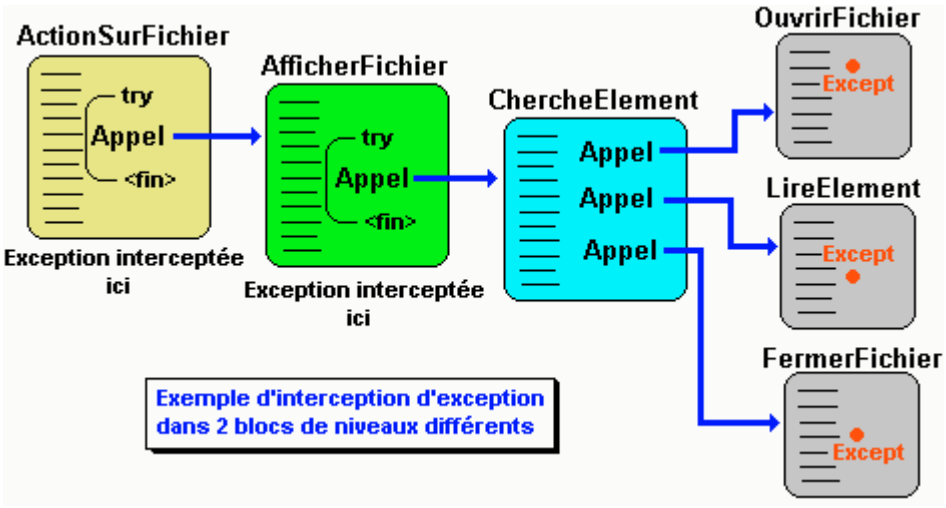
- ***Les blocs lançant éventuellement une exception***

Chaque bloc le plus interne peut lancer (lever) une exception de classe différente et la propage au niveau supérieur :



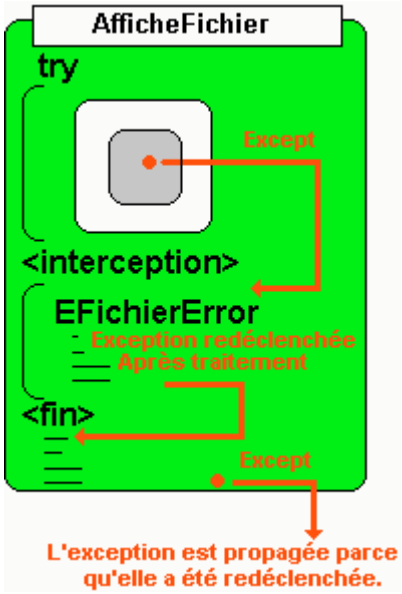
- ***Les blocs interceptant les exceptions***

Nous proposons par exemple d'intercepter les exceptions dans les deux blocs <**ActionsSurFichier**> et <**AfficheFichier**> :



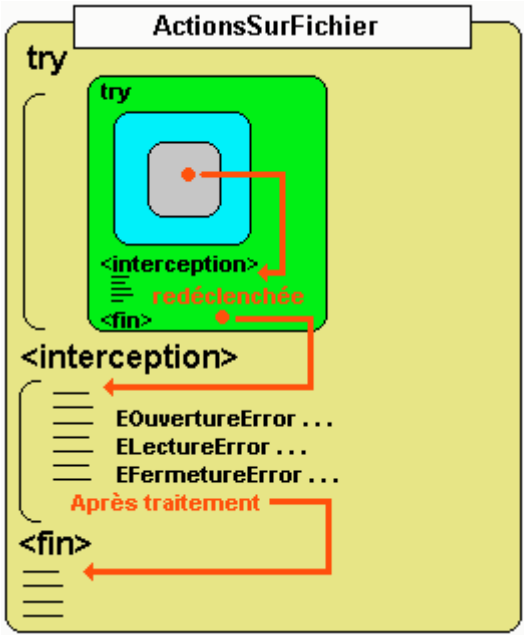
Le bloc <AfficherFichier>

Ce bloc interceptera une exception de type EFichierError, puis la redéclenchera après traitement :



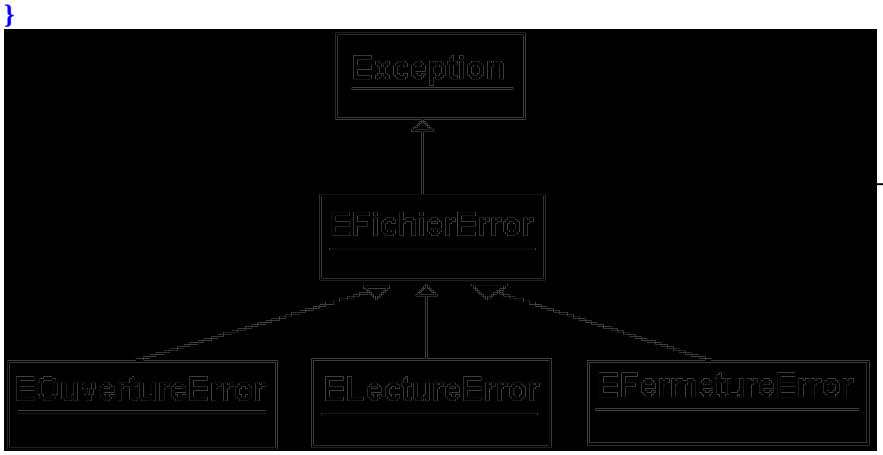
Le bloc <ActionsSurFichier>

Ce bloc interceptera une exception de l'un des trois types EOuvertureError, ELectureError ou EFermetureError :



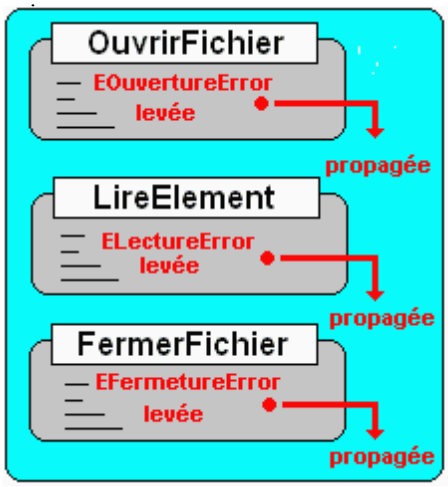
4. Une solution de l'exemple précédent en C#

```
using System ;
namespace PrTests
{
  /* pseudo-Traitement d'un fichier è 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 ) {
  }
}
```



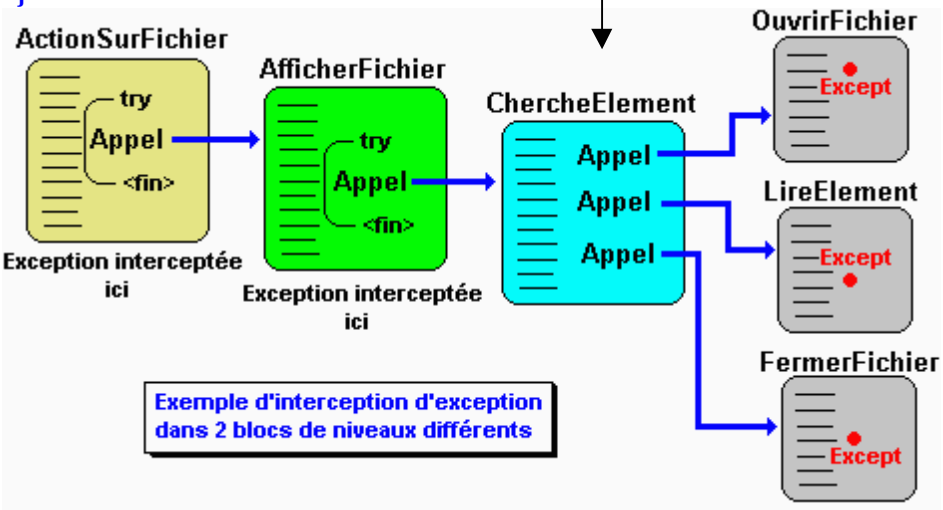

```
//---------------------------------------------------------------------

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.");
 }
```



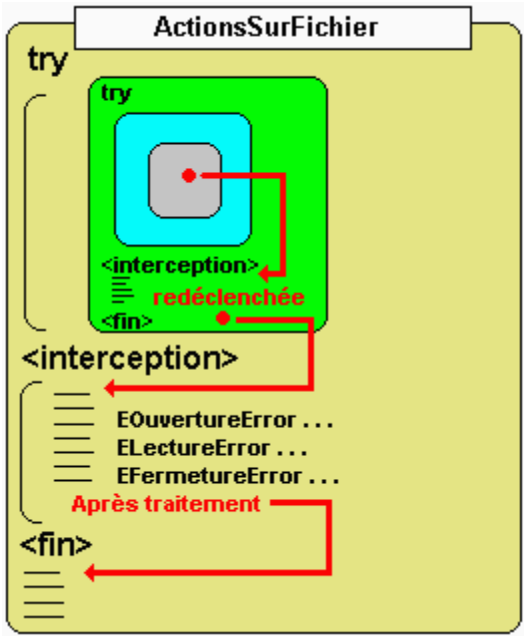

```
void  FermerFichier ( )  {
   System .Console.WriteLine ("  >> Action fermeture...");
   GenererIncident ( 3 );
   System .Console.WriteLine ("  >> Fin fermeture.");
 }
```

```
void  ChercheElement ( )  {
   OuvrirFichier ( );
   LireElement ( );
   FermerFichier ( );
 }
```

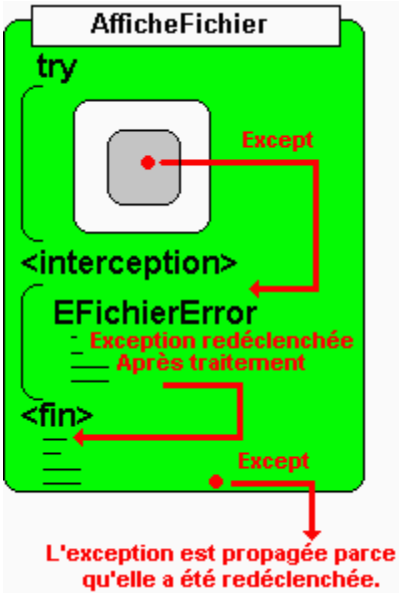



```
//-------------------------------------------------------------------------------------

  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 ( );
   }
  }
 }
```



Algorithmes simples avec

Calcul de la valeur absolue d'un nombre réel ……..p.335
Résolution de l'équation du second degré dans R ……..……..p.337
Calcul des nombres de Armstrong ……..……..p.339
Calcul de nombres parfaits ……..……..……..p.341
Calcul du pgcd de 2 entiers (méthode Euclide) ……..……..p.343
Calcul du pgcd de 2 entiers (méthode Egyptienne) ……..…..……..p.345
Calcul de nombres premiers (boucles **while** et **do**…**while**) **……..…..……..p.347**
Calcul de nombres premiers (boucles **for**) **……..……..p.349**
Calcul du nombre d'or ……..……..p.351
Conjecture de Goldbach ……..……..……..p.353
Méthodes d'opérations sur 8 bits ……..……..……..p.355
Chaînes palindromes 2versions ……..……..……..p.359
Convertir une date numérique en lettres ……..……..……..p.362
Convertir un nombre écrit en chiffres romains ……..……..……..p.364
Tri à bulles tableau d'entiers ……..……..……..p.365
Tri par insertion tableau d'entiers ……..……..p.367
Recherche linéaire dans un tableau non trié ……..……..p.370
Recherche linéaire dans un tableau déjà trié ……..……..p.374
Recherche dichotomique dans un tableau déjà trié ……..……..p.377

*Dans la majorité des exemples de traduction d'algorithmes simples nous reprenons volontairement des exemples déjà traités dans le livre les **fondements de Java**, le lecteur remarquera la similarité du code C# et du code Java pour le même exercice.*

Lorsque cela est le cas nous avons fourni le code sans explication autre que celle de l'énoncé, 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.380
Classe de salariés dans une entreprise fictive ……..p.382
Classe de salariés dans un fichier de l'entreprise …….……...p.393
Construction d'un ensemble de caractères ……..……..p.401
Construction d'un ensemble générique ……..……..……..p.407
Construction d'une IHM de jeu de puzzle …..……...…….p.415

Les exercices utilisent des classes spécifiques au langage C#, si le lecteur veut traduire ces exemples en code Java ou en code Delphi, il doit soit chercher dans les packages Java ou Delphi des classes possédant les mêmes fonctionnalités soit les construire lui-même.

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 , si x ≥ 0

|x| = -x 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 ereurs 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 carré d'un nombre réel :

√Δ 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 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 ;
   if ( b>a) {
          int 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*
Itération
  $n \leftarrow n + 1$;
  $U_n \leftarrow U_{n1} + U_{n2}$;
  $V_n \leftarrow U_n / U_{n1}$;
  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
fin Itération
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) :
       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** :

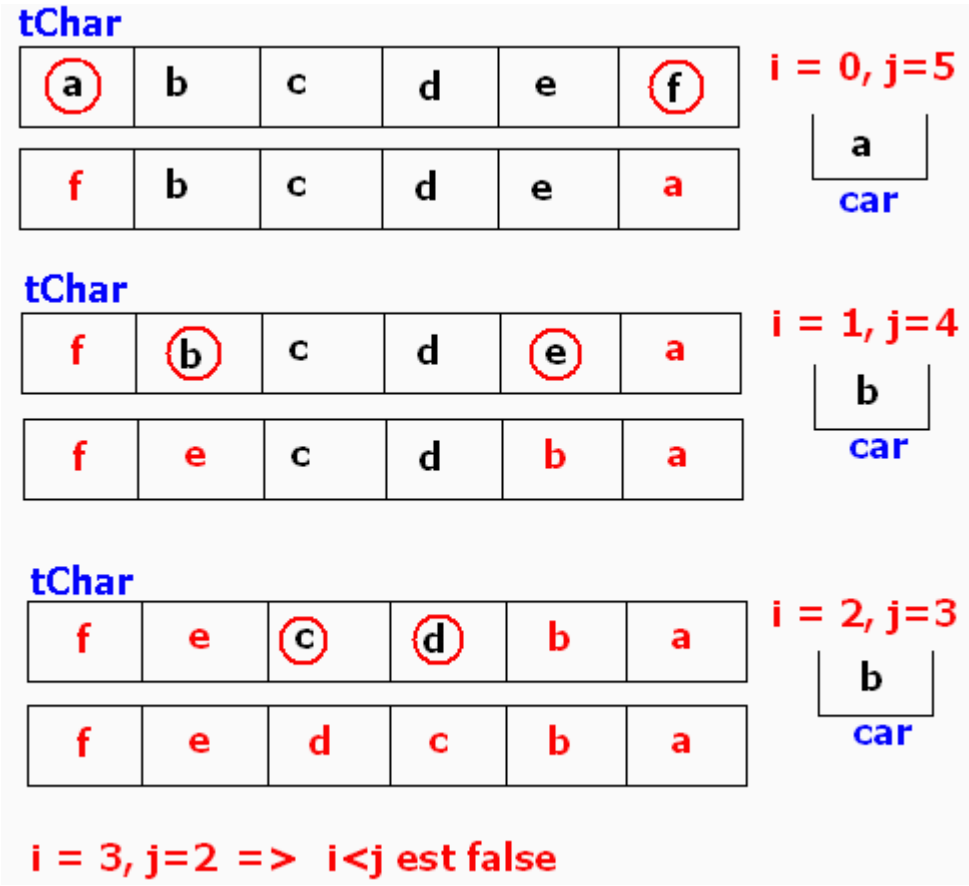
```
static string inverser ( string s )
{
  String strLoc = "";
  for( int i = 0 ; i < s.Length ; i ++ )
   strLoc = s[i] + strLoc ;
  return strLoc ;
}
```

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, diamnche=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 ApplicationTriInsert  {
 static int[ ] table ; // le tableau à trier, par exemple 19 éléments
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 :
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
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 ,
```

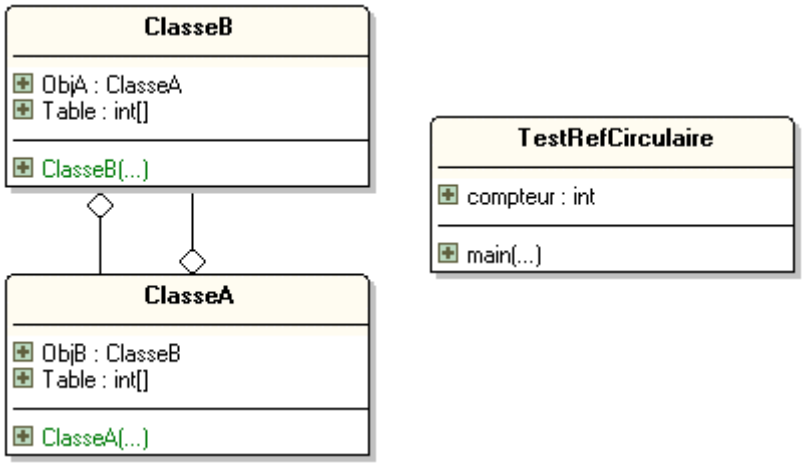
```
                                82 , 69 , 65 , 4 , 95 , 46 , 12 , 87 , 75 };
        table = tableau;
    }
    static void TriInsert ( )
    {
        // sous-programme de Tri par insertion :
        int n = table.Length-1;
        for ( int i = 2; i <= n; i++) {
          int v = table[i];
          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 que 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 :

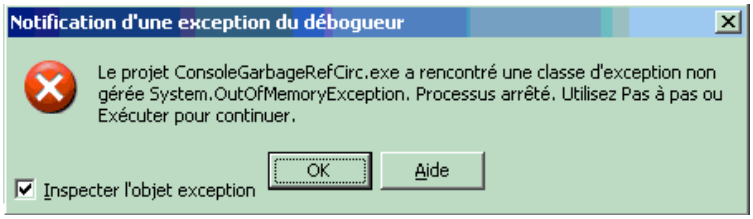
```
using System;

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 :



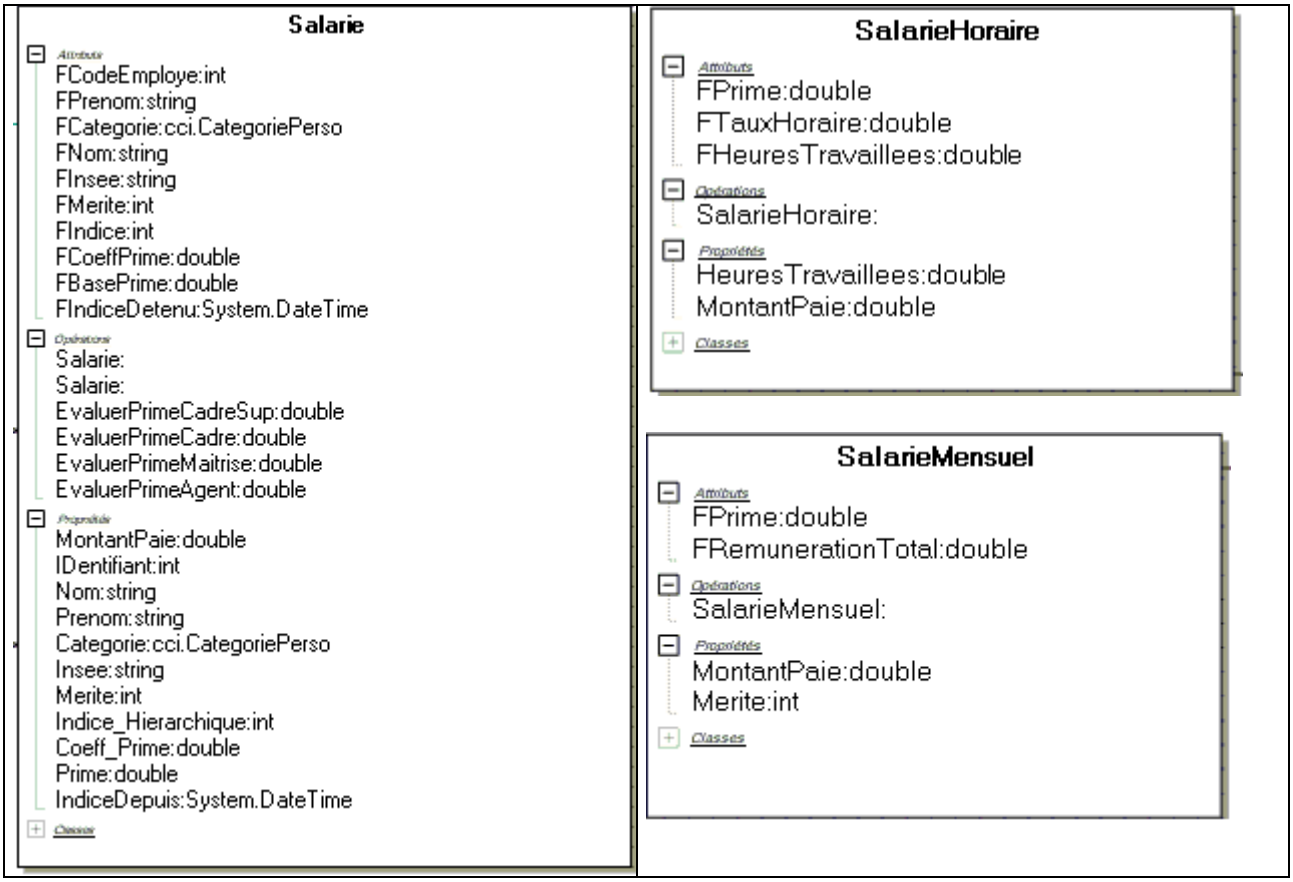
Explications :

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.

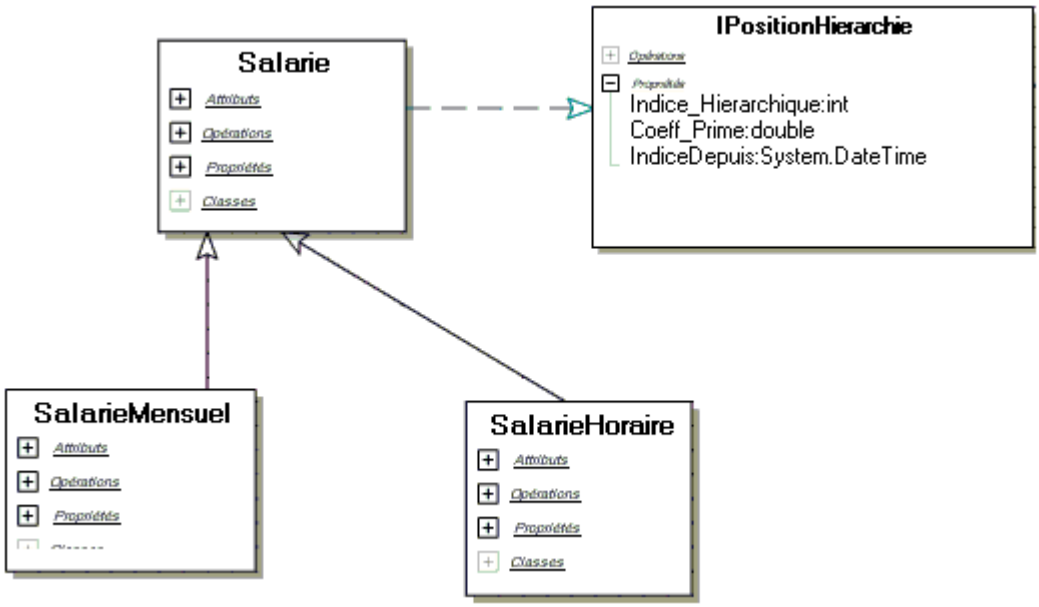
Classes, objet et IHM

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 :

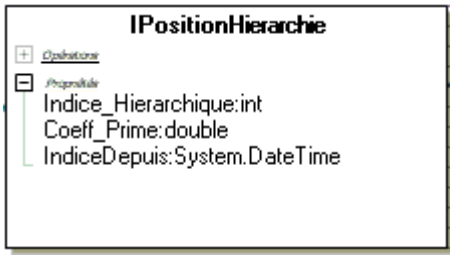


La classe **Salarie** implémente l'interface **IpositionHierarchie** et les classes **SalarieMensuel** et **SalarieHoraire** héritent toutes deux de la classe Salarie :



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 ;
```

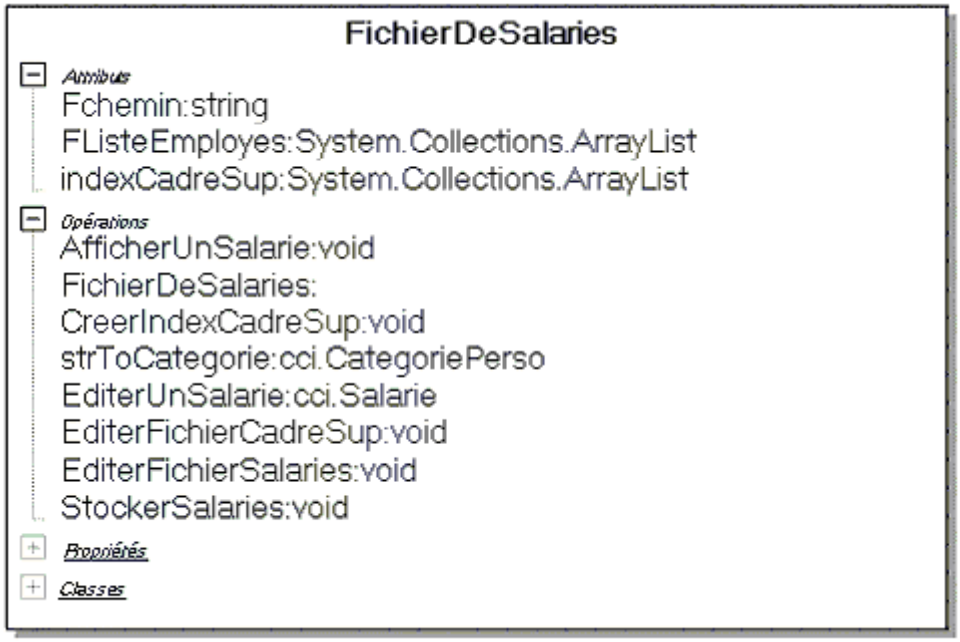
```
  FMerite = Merite ;
  FIndice = Indice ;
  FCoeffPrime = CoeffPrime ;
  FIndiceDetenu = DateTime.Now ;
 switch ( FCategorie )
 {
  case CategoriePerso.Cadre_Sup :
   FBasePrime = 2000 ;
  break;
  case CategoriePerso.Cadre :
   FBasePrime = 1000 ;
  break;
  case CategoriePerso.Maitrise :
   FBasePrime = 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 Ajoute ***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 concatènes, puis on appelle le deuxième constructeur. On 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 Inserer 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** :

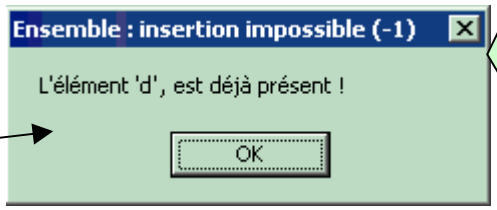
public static DialogResult **Show(** string ***TexteMess*** , string ***caption*** **);**

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).

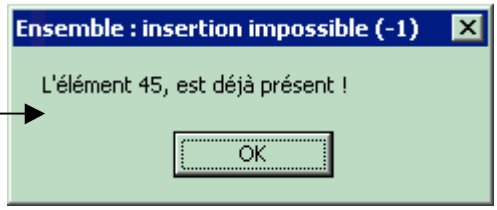
```
public int Add( object value )  {
 // ajoute un élément à l'ensemble sinon -1 ou -2
 if (this.Count==0)
    FTypeElement = value.GetType( );
 If (value.GetType( ) ==TypeElement)  {
  if (!Contains(value) )
   return( List.Add( value ) );
  else  {
   MessageBox.Show("L'élément "+value.ToString()+", 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 );
}
}
```

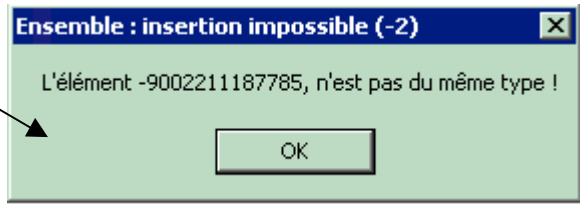
Extrait de code de test où E1, E2 et EnsCar sont des setOfObject	
//-- chargement par ajout élément par élément : E1.Add('a'); E1.Add('b'); E1.Add('c'); E1.Add('d'); E1.Add('e'); //-- chargement par string : E2 = new setOfObject("xyztu#"); EnsCar = E1 + E2 ; foreach (object elt in EnsCar) Console.WriteLine(" "+ 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.

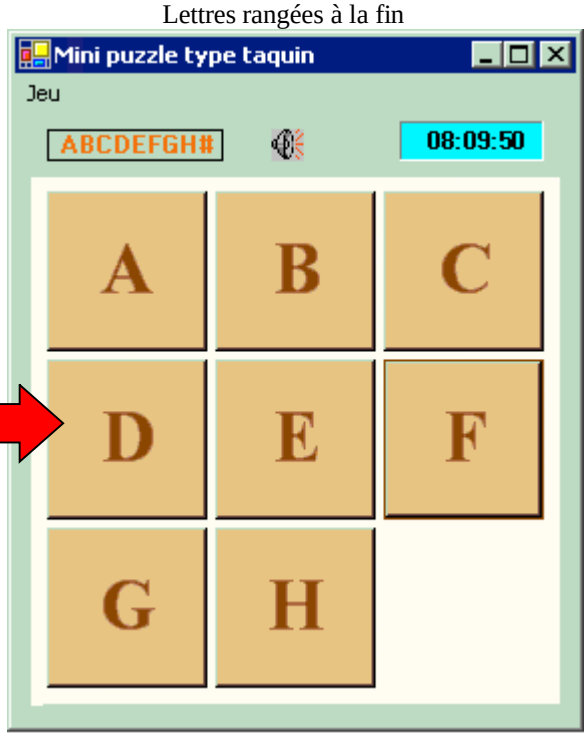
Classes, objet et IHM

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 :

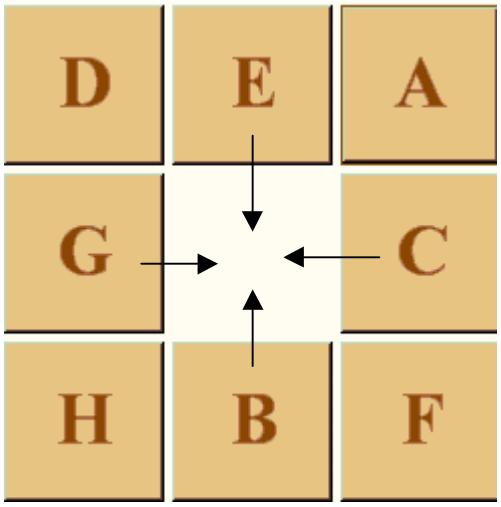
Lettres au départ dans le désordre

Lettres rangées à la fin

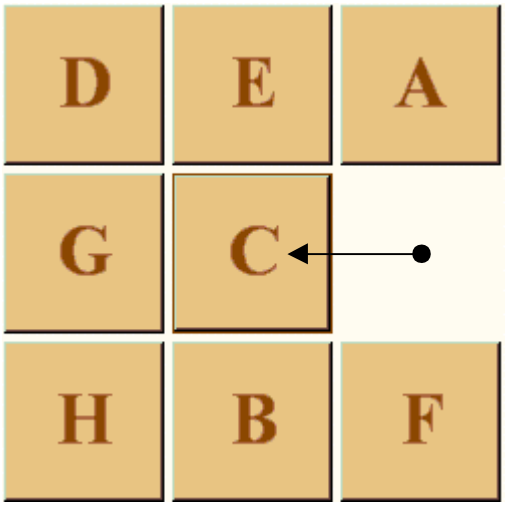


Sur les 9 cases une seule est disponible, le jeu consiste à ne déplacer qu'une seule lettre à la fois et uniquement dans la case restée libre. Par exemple dans la configuration de départ ci-haut seules les lettres G, E, C , B peuvent être déplacées vers la case centrale restée libre.

4 choix de déplacements sont possibles :

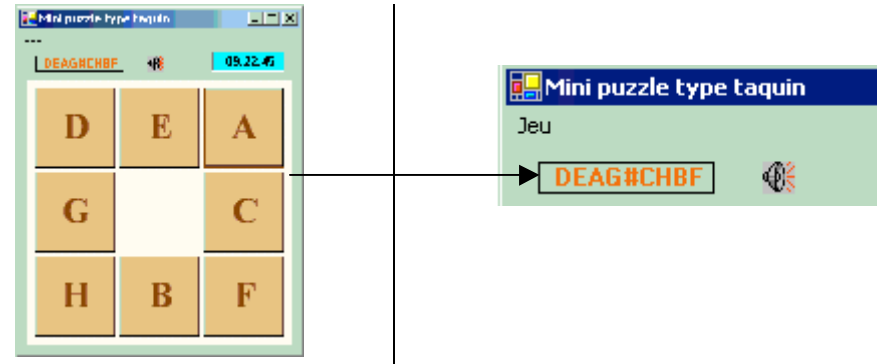


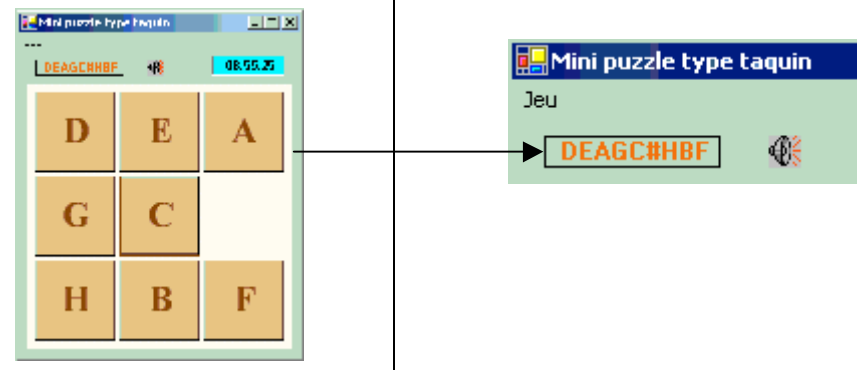
si l'on déplace la lettre C, on obtient :



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).

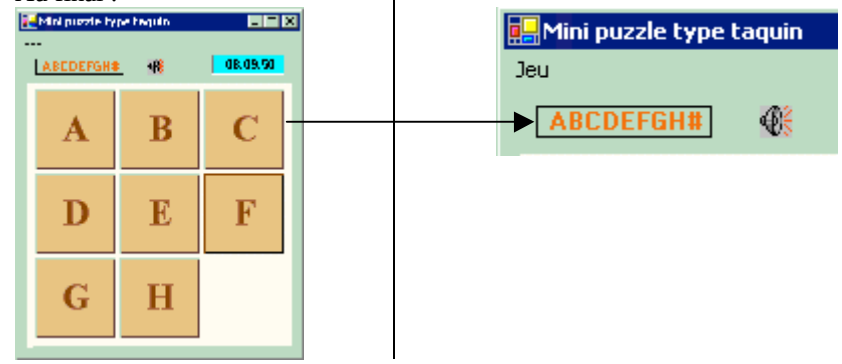
L'IHM affichera la liste des lettres lue depuis le coin supérieur gauche jusqu'au coin inférieur droit du tableau (la case libre sera représentée par un #)



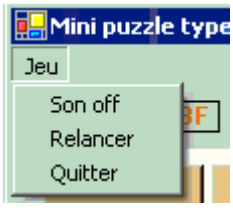


etc …

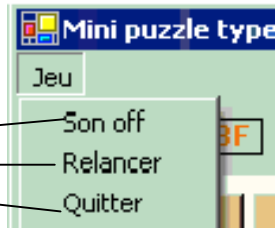
Au final :

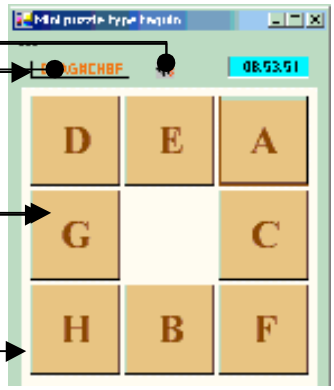


Lorsque le joueur a trouvé la bonne combinaison et rangé les lettres dans le bon ordre, prévoir de lui envoyer une fenêtre de dialogue de félicitation et un menu lui permettant de rejouer ou de quitter le jeu:



```
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( ));
}

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( );
  }
 }
```

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.

```
 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
```

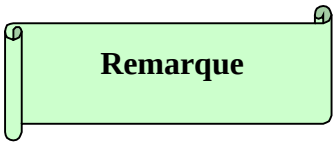
```
//
this.buttonB.BackColor = System.Drawing.Color.Tan;
this.buttonB.Font = new System.Drawing.Font("Times New Roman", 27.75F,
                    System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.buttonB.ForeColor = System.Drawing.Color.SaddleBrown;
this.buttonB.Location = 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.AutoScaleBaseSize = new System.Drawing.Size(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;
 }
}

}
```



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.

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)

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