



# Programmer objet dans .Net Framework avec le langage C# version Orcas



*757 pages de cours + 94 pages d'exercices corrigés*

## **Rm di Scala**

Mise à jour d'octobre 2007

*Pour les mises à jour et les autres cours pdf du même auteur,  
consultez le site suivant : <http://www.discalanet>*

# SOMMAIRE

## Types, opérateurs, instructions

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

## Structures de données de base

Classe String .....	P.73
Tableaux, matrices .....	P.81
Collections, hash, piles, files, listes .....	P.96

## C# est orienté objet

Classes, objets et méthodes .....	P.125
Polymorphisme d'objet .....	P.160
Polymorphisme de méthode .....	P.172
Polymorphisme d'interfaces .....	P.198
Classe de délégation .....	P.223
Traitement des exceptions .....	P.239
Processus et multi-threading .....	P.263

## IHM avec C#

Programmation événementielle .....	P.293
Les événements en C # .....	P.317
Propriétés et indexeurs .....	P.338
Fenêtres et ressources mémoires .....	P.366
Contrôles dans les formulaires .....	P.405
Exceptions comparées à Delphi et java .....	P.429
Données simples flux et fichiers .....	P.433

## Eléments principaux depuis la version C# 2.0

Les generics .....	P.466
Les classes partielles ... ..	P.476
Les méthodes anonymes .....	P.484
TAD de liste, pile, file et classes génériques .....	P.495
TAD d'arbre binaire et classes génériques .....	P.513

Principes des bases de données relationnelles .....	P.554
ADO.Net et données relationnelles .....	P.580
ADO.Net et SQL serveur 2005 .....	P.593

## Programmation web avec ASP.net

Page web écrite uniquement avec des contrôles HTML simples .....	P.636
Page web écrite uniquement avec des contrôles serveurs.....	P.649
Communication entre pages web.....	P.669
Page web et BD avec GridView.....	P.680
Cookie dans une page web .....	P.698
Annexes javascript .....	P.710
Webservices .....	P.735

## Bibliographie .....

## Exercices .....

**94 pages d'exercices avec solutions**

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

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

*A tous ceux qui m'ont envoyé un petit mail me signalant une erreur, une omission ...*

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

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

### **Remerciements :** *(diffusion de la connaissance)*

- A l'université de Tours qui supporte et donne accès à la partie Internet du package pédagogique à partir de sa rubrique "cours en ligne", à partir duquel ce document a été élaboré.

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

### **Remerciements :** *(anticipés)*

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

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

# Introduction à .Net

## C#.net

---

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

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

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

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

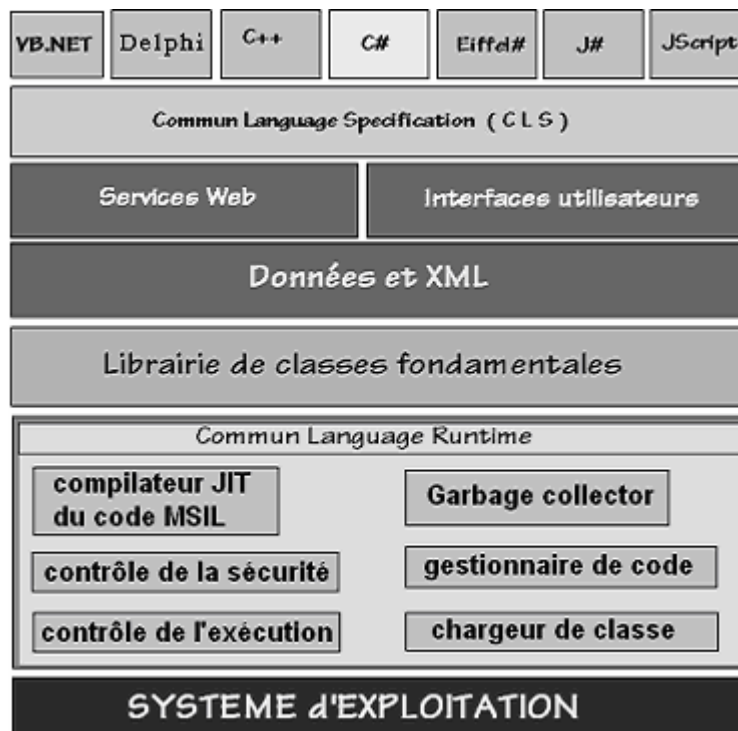
Les versions gratuites soit "Express" de Microsoft, soit l'environnement complet en version professionnelle de Visual Studio **.NET**, soit sharpDevelop de l'open source démocratisent les outils de programmation.


# Les outils élémentaires

## C#.net

### 1. La plate forme .NET Framework

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




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

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




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

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

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



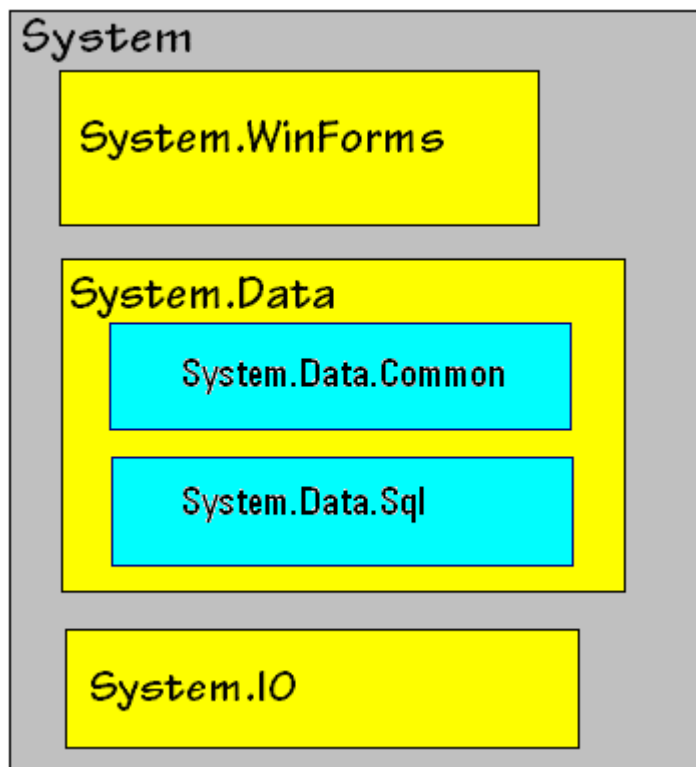
Les données sont accédées dans le cas des services Web à travers les protocoles qui sont des standards de l'industrie : HTTP, XML et SOAP.

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




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

La librairie de classe de **.NET Framework** est organisée en nom d'espace hiérarchisés, 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**" se déclare comme "**System.Data.DataSet**".
- La classe **Console** qui se trouve dans l'espace de noms "**System**" se déclare comme "**System.Console**".
- La classe **DataAdapter** qui se trouve dans l'espace de noms "**System.Data.Common**" se déclare comme "**System.Data.Common.DataAdapter**".

 **La quatrième couche** forme l'environnement d'exécution commun (**CLR** ou Common Language Runtime) de tous les programmes s'exécutant dans l'environnement .NET. Le **CLR** exécute un bytecode écrit dans un langage intermédiaire (**MSIL** ou **Microsoft Intermediate Language**)

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 nécessaire 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'implémentation 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.**

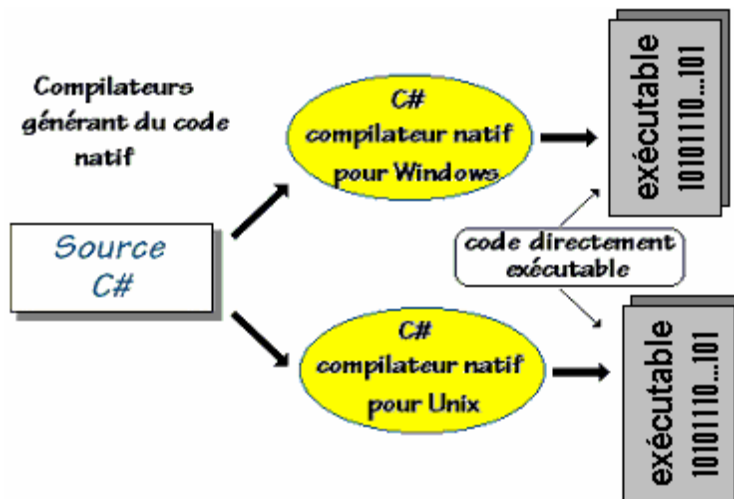
Beaucoup de langages possèdent pour une plate-forme fixée des interpréteurs ou des compilateurs, moins possèdent une p-machine, Java de Sun est l'un de ces langages. Tous les langages de la plateforme .NET fonctionnent selon ce principe, **C# conçu par microsoft en est le dernier**, un programme C# compilé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





Programme source C# : xxx.cs

Programme exécutable sous Windows : xxx.exe (code natif processeur)

## Bytecode ou langage intermédiaire

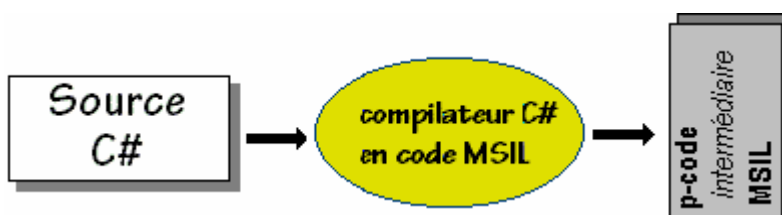
La compilation en bytecode (ou pseudo-code ou p-code ou code intermédiaire) est semblable à l'idée du p-code de N.Wirth pour obtenir un portage multi plate-formes du pascal. Le compilateur C# de **.NET Framework** traduit le programme source xxx.cs en un code intermédiaire indépendant de toute machine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



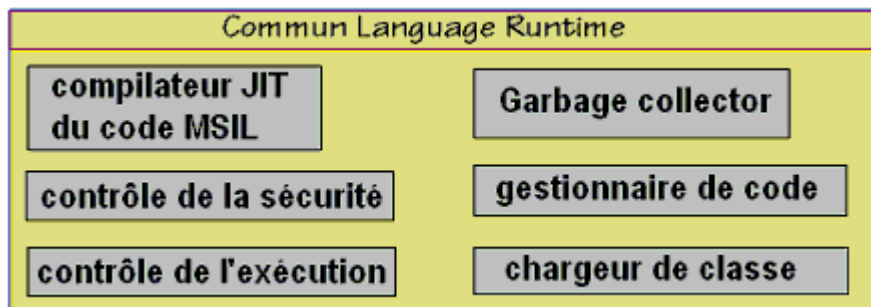
Programme source C# : Exemple.cs

Programme exécutable sous .NET : Exemple.exe (code portable IL)

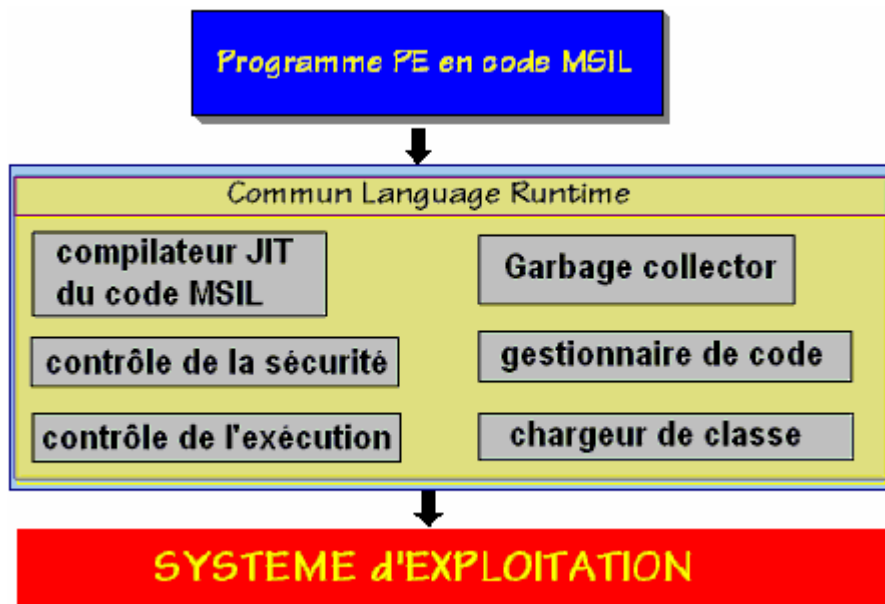
## 2. L'environnement d'exécution du CLR

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

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



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



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

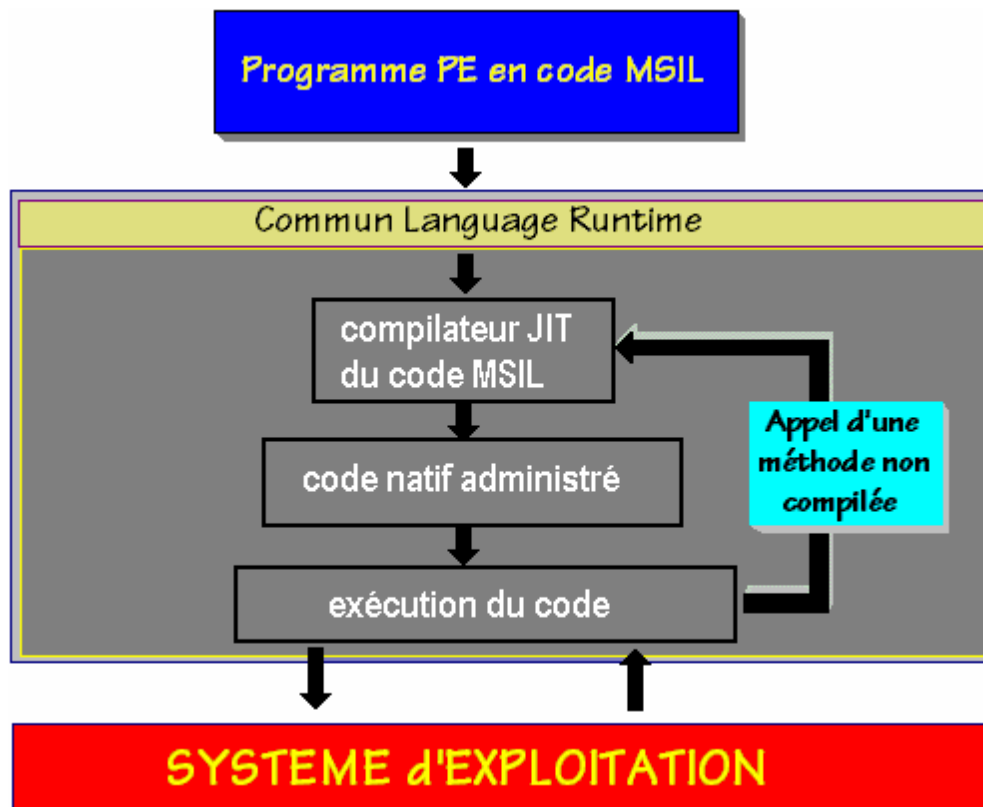
## La compilation JIT progressive

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

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

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

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



## La compilation AOT

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

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

# Les éléments de base

## C#.net

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).

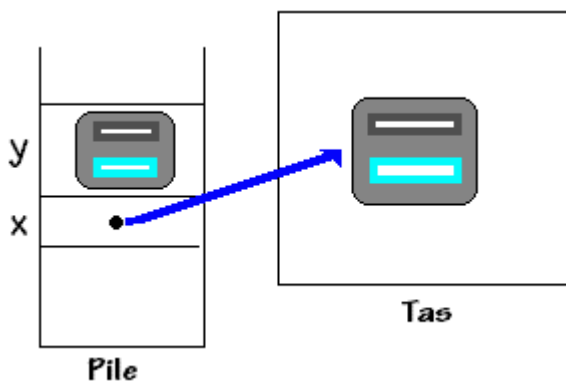
### 1. Les types valeurs du CLS dans .NET Framework

#### 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'une 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 :

<p><b>Déclaration de classe-structure :</b></p> <pre> <b>struct</b> StructAmoi {     <b>int</b> b;     <b>void</b> meth(<b>int</b> a){         b = 1000+a;     } }                     </pre>	<p><b>instanciation :</b></p> <pre> StructAmoi y = <b>new</b> StructAmoi ( ) ;                     </pre>
<p><b>Déclaration de classe :</b></p> <pre> <b>class</b> ClassAmoi {     <b>int</b> b;     <b>void</b> meth(<b>int</b> a) {         b = 1000+a;     } }                     </pre>	<p><b>instanciation :</b></p> <pre> ClassAmoi x = <b>new</b> ClassAmoi ( ) ;                     </pre>

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

Exemple ci-dessous 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" ;
                    
```

<i>Classe-structure</i>	<i>intervalle de variation</i>	<i>nombre de bits</i>
<b>Boolean</b>	<b>false , true</b>	1 bit

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

### Compatibilité des types de .NET Framework

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

[Visual Basic]

**Public Structure** Int32

**Implements** IComparable, IFormattable, IConvertible

[C#]

**public struct** Int32 : IComparable, IFormattable, IConvertible

[C++]

**public \_\_value struct** Int32 : **public** IComparable, IFormattable,  
IConvertible

[J#]

**public class** Int32 **extends** System.ValueType **implements** System.IComparable,  
System.IFormattable, System.IConvertible

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

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

<i>Types valeurs C#</i>	<i>Classe-structure de .NET Framework</i>	<i>nombre de bits</i>
<b>bool</b>	Boolean	1 bit
<b>sbyte</b>	SByte	8 bits
<b>byte</b>	Byte	8 bits

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

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

### Remarque importante

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

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

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

### Exemples :

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

## Transtypage opérateur ( )

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

### Transtypage implicite en C# :

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

### Transtypage explicite en C# :

`int x;`

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

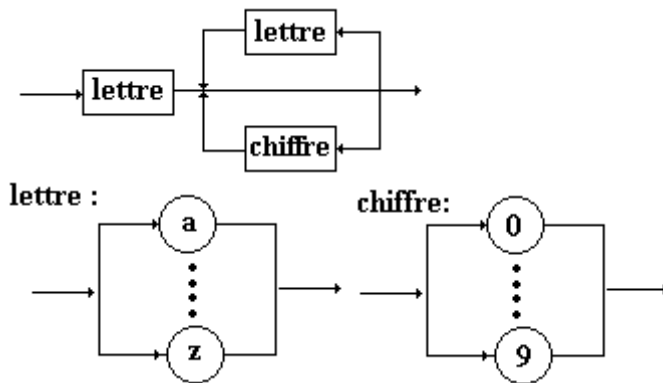
- Tous les types élémentaires peuvent être transtypés à l'exception du type **bool** qui ne peut pas être converti en un autre type (différence avec le C).
- Les conversions **peuvent être restrictives** quant au résultat; par exemple le transtypage du réel 5.27e-2 en entier ( `x = (int)5.27e-2` ) mettra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

## 3. Variables, valeurs, constantes en C#

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

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

## Identificateur C# :



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

### Exemples de déclaration de variables :

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

etc ...

Exemples d'affectation de valeurs à ces variables :

<i>Affectation</i>	<i>Déclaration avec initialisation</i>
<pre>Bonjour = 2587 ; Valeur = -123.5687 UnCar = 'K' ; Test = false ;</pre>	<pre>int Bonjour = 2587 ; float Valeur = -123.5687 char UnCar = 'K' ; bool Test = false ;</pre>

### 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 qualificatifs de déclaration qui se rajoutent devant les autres qualificatifs 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 :*

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

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*

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

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 :

<code>int k = 5 , n ; n = k++ ;</code>	<code>n = 5</code>	<code>k = 6</code>
--	--------------------	--------------------

#### Exemple 2 :

<code>int k = 5 , n ; n = k++ - k ;</code>	<code>n = -1</code>	<code>k = 6</code>
--	---------------------	--------------------

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 :

<code>int k = 5 , n ; n = k - k++ ;</code>	<code>n = 0</code>	<code>k = 6</code>
--	--------------------	--------------------

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 incrémentement 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 ;`

<code>nbr1 = 10 ; v = nbr1++</code>	<code>v = 10</code>	<code>nbr1 = 11</code>
<code>nbr1 = 10 ; z = ~ nbr1 ;</code>	<code>z = -11</code>	<code>nbr1 = 10</code>
<code>nbr1 = 10 ; t = ~ nbr1 ++ ;</code>	<code>t = -11</code>	<code>nbr1 = 11</code>
<code>nbr1 = 10 ; u = ~ (nbr1 ++ ) ;</code>	<code>u = -11</code>	<code>nbr1 = 11</code>

La notation " $\sim \text{nbr1} ++$ " est refusée par C# comme pour Java.

Remarquons que les expressions " $\sim \text{nbr1} ++$ " et " $\sim (\text{nbr1} ++)$ " produisent les mêmes effets, ce qui est logique puisque lorsque deux opérateurs (ici  $\sim$  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.

#### Exemple 1 :

`int k = 5, n ;`

<code>n = ++k ;</code>	<code>n = 6</code>	<code>k = 6</code>
------------------------	--------------------	--------------------

#### Exemple 2 :

<code>int k = 5, n ; n = ++k - k ;</code>	<code>n = 0</code>	<code>k = 6</code>
---	--------------------	--------------------

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 :

<code>int k = 5, n ; n = k - ++k ;</code>	<code>n = -1</code>	<code>k = 6</code>
---	---------------------	--------------------

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.

#### Exemple 1 :

`int k = 5, n ;`

<code>n = k-- ;</code>	<code>n = 5</code>	<code>k = 4</code>
------------------------	--------------------	--------------------

## pré-décrémentation : --k

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

*Exemple1 :*

`int k = 5, n ;`

<code>n = --k ;</code>	<code>n = 4</code>	<code>k = 4</code>
------------------------	--------------------	--------------------

*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 opérands.

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	<code>X is int</code> ; <code>if ( x is Object )</code> etc...

## 4. Opérateurs booléens

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

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

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

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

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

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

p	q	! p	p & q	p ^ q	p   q
V	V	F	V	F	V
V	F	F	F	V	V
F	V	V	F	V	V
F	F	V	F	F	F

**Remarque :**

$\forall p \in \{ V, F \}, \forall q \in \{ V, F \}, p \ \& \ q$  est toujours évalué en entier ( p et q sont toujours évalués).  
 $\forall p \in \{ V, F \}, \forall q \in \{ V, F \}, p \ | \ q$  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 différencient de leurs originaux **&** et **|** par leur mode d'exécution optimisé (application de théorèmes de l'algèbre de boole) :

**L'opérateur et optimisé : &&**

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

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 « et » noté **&&** qui a **la même table de vérité** que l'opérateur **&** mais qui applique ce théorème.

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

**L'opérateur ou optimisé : ||**

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

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 `|`.

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

**En résumé:**

Opérateur	priorité	action	exemples
<b>!</b>	1	<b>non</b> booléen	<code>!(5 &lt; 2)</code> ; <code>!(x+1 &lt; 3)</code> ; etc...
<b>&amp;</b>	7	<b>et</b> booléen complet	<code>(5 == 2) &amp; (x+1 &lt; 3)</code> ; etc...
<b> </b>	9	<b>ou</b> booléen complet	<code>(5 != 2)   (x+1 &gt;= 3)</code> ; etc...
<b>&amp;&amp;</b>	10	<b>et</b> booléen optimisé	<code>(5 == 2) &amp;&amp; (x+1 &lt; 3)</code> ; etc...
<b>  </b>	11	<b>ou</b> booléen optimisé	<code>(5 != 2)    (x+1 &gt;= 3)</code> ; 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
<code>~</code>	1	<b>complémenter</b> les bits	<code>~a</code> ; <code>~(a-b)</code> ; <code>~7</code> (unaire)
<code>&lt;&lt;</code>	4	<b>décalage</b> gauche	<code>x &lt;&lt; 3</code> ; <code>(a+2) &lt;&lt; k</code> ; <code>-5 &lt;&lt; 2</code> ;
<code>&gt;&gt;</code>	4	<b>décalage</b> droite avec signe	<code>x &gt;&gt; 3</code> ; <code>(a+2) &gt;&gt; k</code> ; <code>-5 &gt;&gt; 2</code> ;
<code>&amp;</code>	7	<b>et</b> booléen bit à bit	<code>x &amp; 3</code> ; <code>(a+2) &amp; k</code> ; <code>-5 &amp; 2</code> ;
<code>^</code>	8	<b>ou exclusif xor</b> bit à bit	<code>x ^ 3</code> ; <code>(a+2) ^ k</code> ; <code>-5 ^ 2</code> ;
<code> </code>	9	<b>ou</b> booléen bit à bit	<code>x   3</code> ; <code>(a+2)   k</code> ; <code>-5   2</code> ;

Les tables de vérités des opérateurs `&`, `|` et celle du ou exclusif `^` au niveau du bit sont identiques aux tables de vérité 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

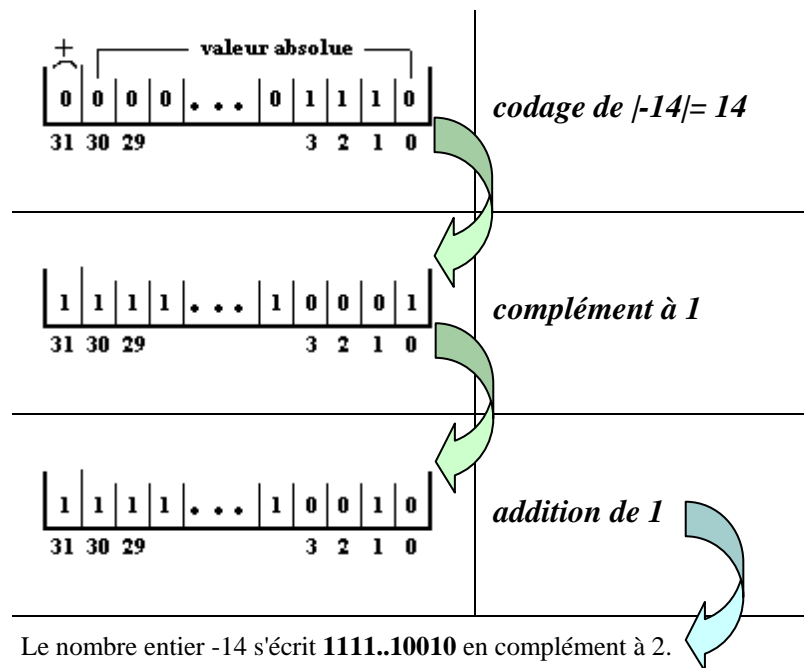
Afin de bien comprendre ces opérateurs, le lecteur doit bien connaître les différents codages des entiers en machine (binaire pur, binaire signé, complément à deux) car les entiers C# sont codés en complément à deux et la manipulation bit à bit nécessite une bonne compréhension de ce codage.

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)



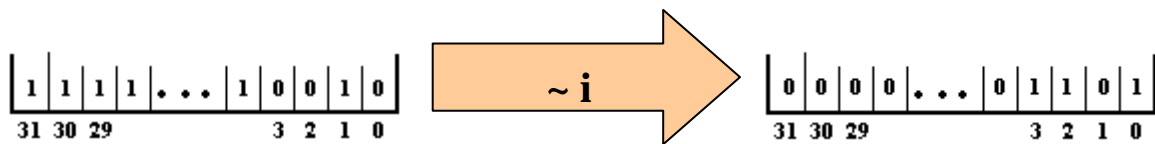
Soient la déclaration C# suivante :

```
int i = -14, j;
```

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

- Etude de l'instruction :  $j = \sim i$

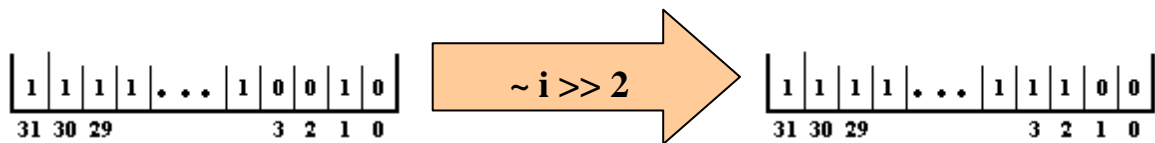
```
j = ~ i ; // complémentation des bits de i
```



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 :  $j = i \gg 2$

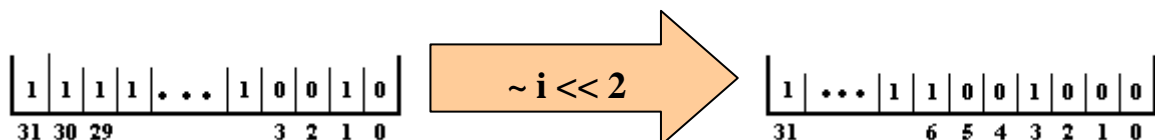
```
j = i >> 2 ; // décalage avec signe de 2 bits vers la droite
```



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 :  $j = i \ll 2$

```
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 qui 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 programme :

```
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 programme :

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

## Exemples opérateurs bit level - Décalages

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

### Résultats d'exécution de ce programme :

```
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 programme :

```
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

## C#.net

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

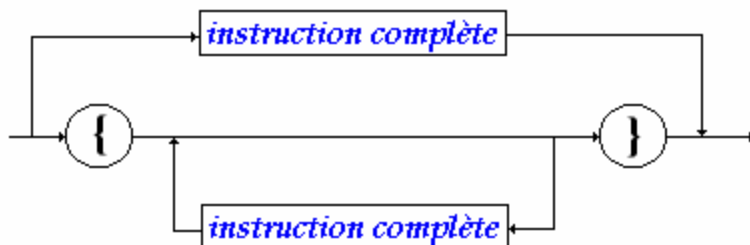
### 1 - les instructions de bloc

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 :*

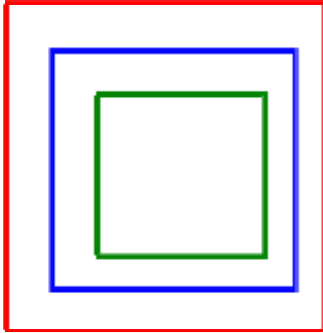
L'élément syntaxique



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 :*

<pre>int a, b = 12; { int x, y = 8 ;   { int z =12;     x = z ;     a = x + 1 ;     { int u = 1 ;       y = u - b ;     }   } }</pre>	 <p><i>schéma d'imbrication des 3 blocs</i></p>
---	---

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
<code>int a , b = 56 ;</code>	<b>a = ???</b>	<b>b = 56</b>
<code>a = (b = 12)+8 ;</code>	<b>a = 20</b>	<b>b = 12</b>

### 3 - Raccourcis et opérateurs d'affectation

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

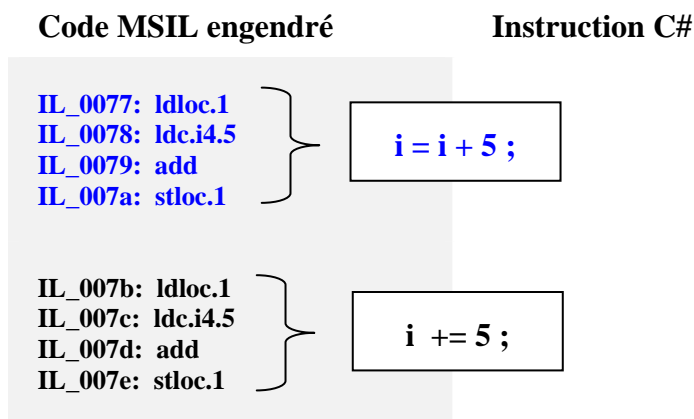
{ +, -, \*, /, %, <<, >>, >>>, &, |, ^ },

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
```

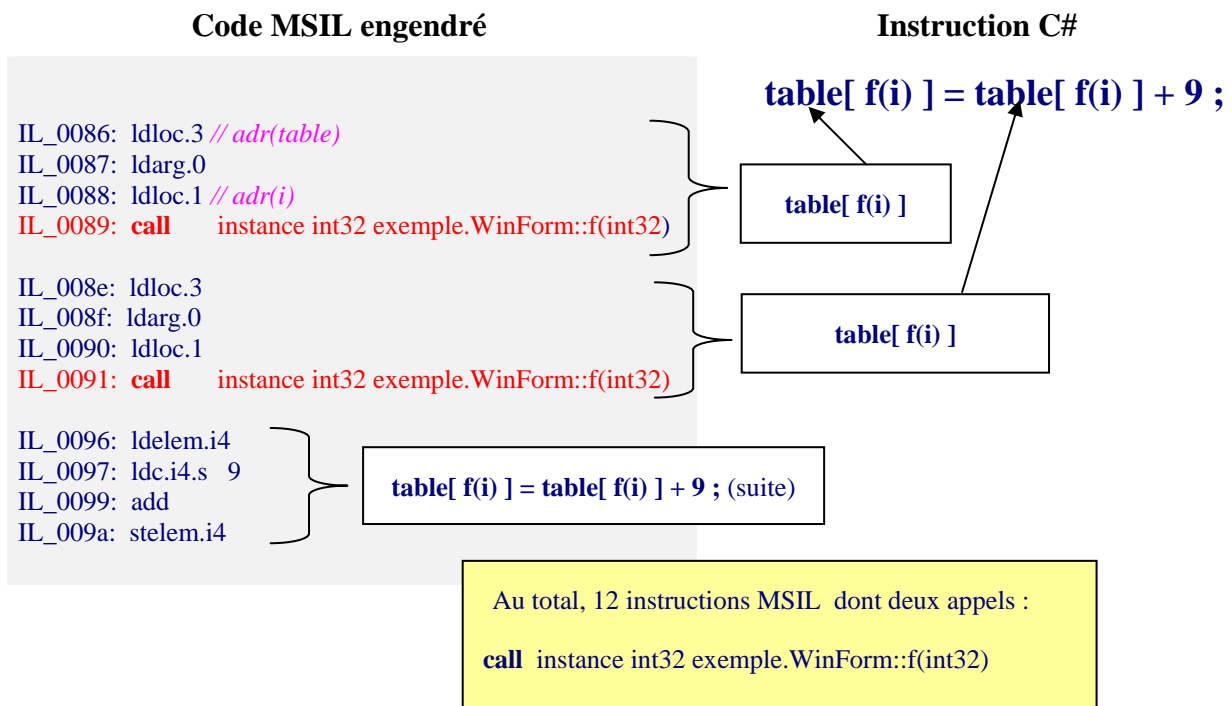
*simulation d'exécution C# :*

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

**Remarques :**

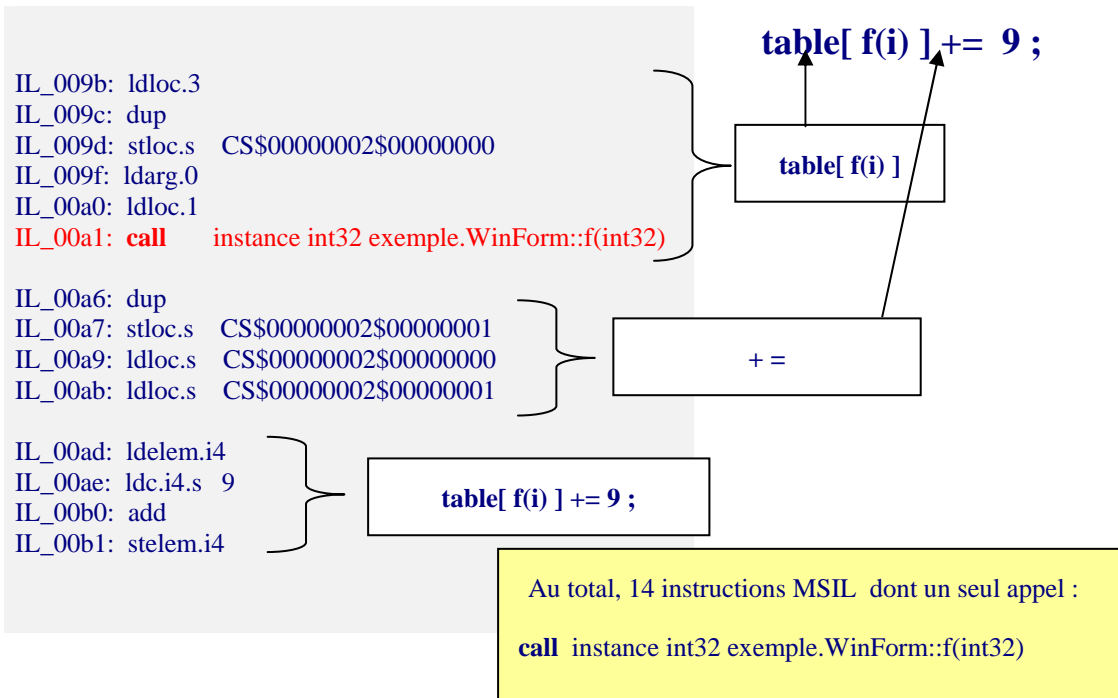
- *Cas d'une optimisation intéressante dans l'instruction suivante :  
table[ f(a) ] = table[ f(a) ] + x ; // où f(a) est un appel à la fonction f qui serait longue à calculer.*
- *Si l'on réécrit l'instruction précédente avec l'opérateur += :  
table[ f(a) ] += x ; // l'appel à f(a) n'est effectué qu'une 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#



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.WinForms::f(int32) >**, est significativement long devant les temps d'exécution des opérations **ldloc** et **stloc**.

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

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

**x = f(i) ;**

**table[ x ] = table[ x ] + 9 ;**

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

### Info MSIL :

**ldloc** : Charge la variable locale à un index spécifique dans la pile d'évaluation.

**stloc** : Dépille la pile d'évaluation et la stocke dans la liste de variables locales à un index spécifié.

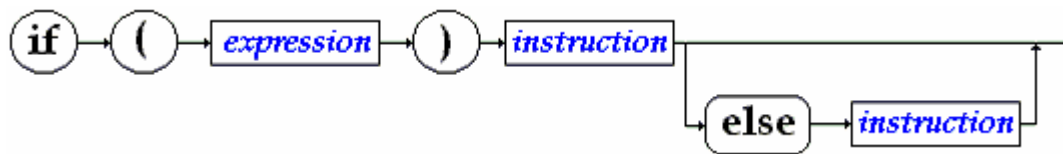


# Les instructions conditionnelles

## C#.net

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

## 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 résout l'ambiguïté 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#
<pre>if Expr1 then begin   if Expr2 then InstrA end else InstrB</pre>	<pre>if ( Expr1 ) {   if ( Expr2 ) InstrA ; } else InstrB</pre>

## 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, booléenne etc...) renvoyant une valeur de type quelconque.

### Sémantique :

Exemple :

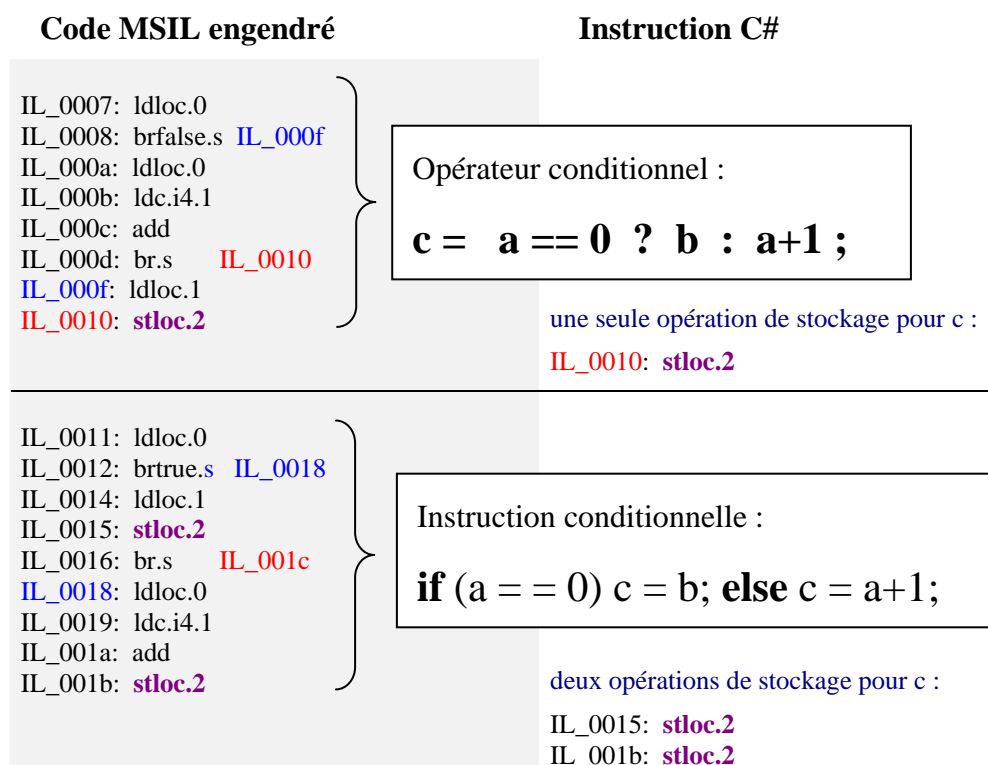
```
int a,b,c ;
c = a == 0 ? b : a+1 ;
```

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.*

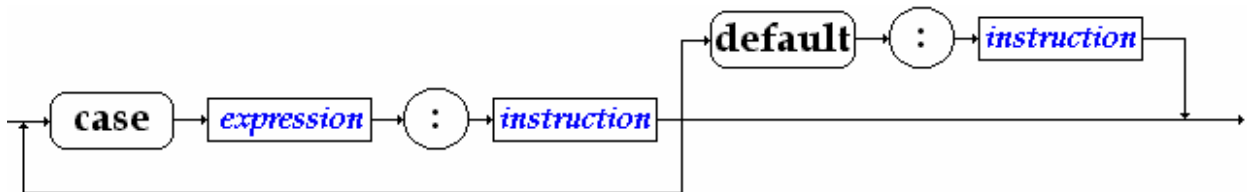
### 3 - l'opérateur switch...case

**Syntaxe :**

*switch :*



*bloc switch :*



**Sémantique :**

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

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

### Exemple de switch..case..break

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

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
<pre> int x = 10;  switch (x+1) { case 11 : System.Console.WriteLine ("&gt;&gt; case 11");   break;   case 12 : System.Console.WriteLine ("&gt;&gt; case 12");   break;   default : System.Console.WriteLine ("&gt;&gt; default");   break; } </pre>	>> case 11
<pre> int x = 11;  switch (x+1) { case 11 : System.Console.WriteLine ("&gt;&gt; case 11");   break;   case 12 : System.Console.WriteLine ("&gt;&gt; case 12");   break;   default : System.Console.WriteLine ("&gt;&gt; default");   break; } </pre>	>> 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 :*

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

Bien que la syntaxe du **switch ...break** soit plus contraignante que celle du **case...of** de Delphi, le fait que cette instruction apporte comme 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

## C#.net

### 1 - l'instruction while

#### Syntaxe :



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

#### Sémantique :

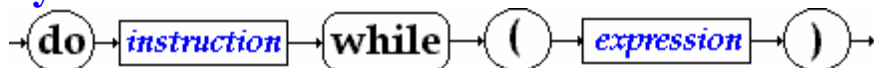
Identique à celle du pascal (instruction algorithmique **tantque .. faire .. ftant**) avec le même défaut de fermeture de la boucle.

#### Exemple de boucle while

Pascal-Delphi	C#
<b>while</b> Expr <b>do</b> Instr	<b>while</b> ( Expr ) Instr ;
<b>while</b> Expr <b>do</b> <b>begin</b> InstrA ; InstrB ; ... <b>end</b>	<b>while</b> ( 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 **jusqu'à non** Expr.

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

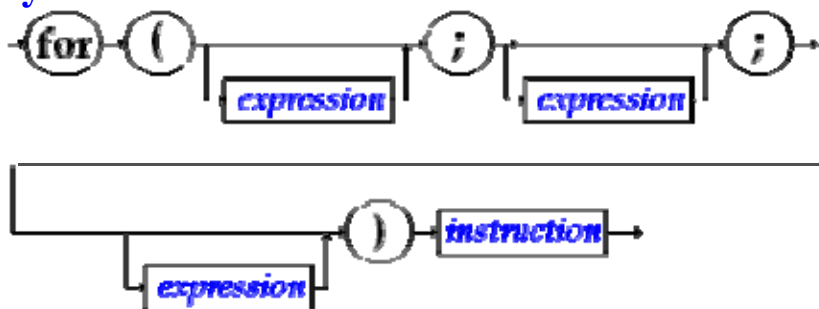
**do** Instr **while** ( Expr )  $\Leftrightarrow$  Instr ; **while** ( Expr ) Instr

### Exemple de boucle do...while

Pascal-Delphi	C#
<pre>repeat   InstrA ;   InstrB ; ... until not Expr</pre>	<pre>do {   InstrA ;   InstrB ; ... } while ( Expr )</pre>

## 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 forme 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... f**pour, elle est toutefois plus puissante que cette dernière.

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

<b>for</b> ( <b>Expr1</b> ; <b>Expr2</b> ; <b>Expr3</b> ) Instr	<b>Expr1</b> ; <b>while</b> ( <b>Expr2</b> ) { Instr ; <b>Expr3</b> }
---	---

(\*)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#
<b>for</b> i:=1 to 10 <b>do</b> <b>begin</b> InstrA ; InstrB ; ... <b>end</b>	<b>for</b> ( <b>i = 1; i&lt;=10; i++</b> ) { InstrA ; InstrB ; ... } }
<b>i := 10; k := i;</b> <b>while</b> (i>-450) <b>do</b> <b>begin</b> InstrA ; InstrB ; ... <b>k := k+i;</b> <b>i := i-15;</b> <b>end</b>	<b>for</b> ( <b>i = 10, k = i ; i&gt;-450 ; k += i , i -= 15</b> ) { InstrA ; InstrB ; ... } }
<b>i := n;</b> <b>while</b> i<>1 <b>do</b> <b>if</b> i mod 2 = 0 <b>then</b> i := i div 2 <b>else</b> i := i+1	<b>int</b> i, j ; <b>for</b> ( <b>i = n, j ; i != 1 ; j = i % 2 == 0 ? i/=2 : i++</b> ); <i>// pas de corps de boucle !</i>

- 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'incréméntation séquentiel étant de 1.
- Le second exemple montre une boucle toujours contrôlée par une variable "i", mais dont le pas de décréméntation séquentiel est de -15.
- Le troisième 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 C#.net

## 1 - l'instruction d'interruption break

### Syntaxe :

→ **break** →

### Sémantique :

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

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

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

### Exemple d'utilisation du break dans un for :

*(recherche séquentielle dans un tableau)*

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

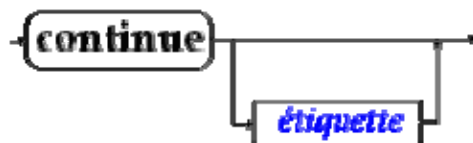
### Explications

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

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

## 2 - l'instruction de rebouclage continue

### Syntaxe :



### Sémantique :

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

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

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

### Exemple d'utilisation du continue dans un for :

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

### Explications

Rappelons qu'un **for** s'écrit généralement :

**for** ( **Expr1** ; **Expr2** ; **Expr3** ) Instr

L'instruction **continue** présente dans une telle boucle **for** s'effectue ainsi :

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

- réexécution du corps de boucle.

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

Le déroulement est alors le suivant :

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

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

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

## Attention

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

<b>for</b> ( <b>Expr1</b> ; <b>Expr2</b> ; <b>Expr3</b> ) Instr	<b>Expr1</b> ; <b>while</b> ( <b>Expr2</b> ) { Instr ; <b>Expr3</b> }
---	---

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 :

<b>for</b> ( <b>i = 0, n = 0 ; i&lt;8 ; i++ , k = 2*n</b> ) { <b>if</b> ( <code>ta[i] == 0</code> ) <b>continue</b> ; <code>tb[n] = ta[i];</code> <code>n++;</code> }	<b>i = 0; n = 0 ;</b> <b>while</b> ( <b>i&lt;8</b> ) { <b>if</b> ( <code>ta[i] == 0</code> ) <b>continue</b> ; <code>tb[n] = ta[i];</code> <code>n++;</code> <b>i++ ; k = 2*n;</b> }
---	--

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 :

<pre><b>for</b> ( <b>i = 0, n = 0 ; i&lt;8 ; i++ , k = 2*n</b> ) { <b>if</b> ( ta[i] == 0 ) <b>continue</b> ;   tb[n] = ta[i];   n++; }</pre>	<pre><b>i = 0; n = 0 ;</b> <b>while</b> ( <b>i&lt;8</b> ) { <b>if</b> ( ta[i] == 0 )   { <b>i++ ; k = 2*n</b>;     <b>continue</b> ;   }   tb[n] = ta[i];   n++;   <b>i++ ; k = 2*n</b>; }</pre>
---	--

# Classes avec méthodes static

## C#.net

---

### Une classe suffit

### Les méthodes sont des fonctions

### Transmission des paramètres en C# (plus riche qu'en java)

### Visibilité des variables

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

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

## 1 - Une classe suffit

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

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

## Syntaxe d'une classe exécutable

*Exemple1 de classe minimale :*

```
class Exemple1 { }
```

Cette classe ne fait rien et ne produit rien.

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

*Exemple2 de squelette d'une classe minimale exécutable :*

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

*Exemple3 trivial d'une classe minimale exécutable :*

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

## Exemples d'applications à une seule classe

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

*Exemple1*

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

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



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

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

```
tableau avant : abcdef
tableau après : fedcba
```

### Exemple2

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

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

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

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

### Conseil de travail :

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

## 2 - Les méthodes sont des fonctions

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

### Méthode élémentaire de classe

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

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

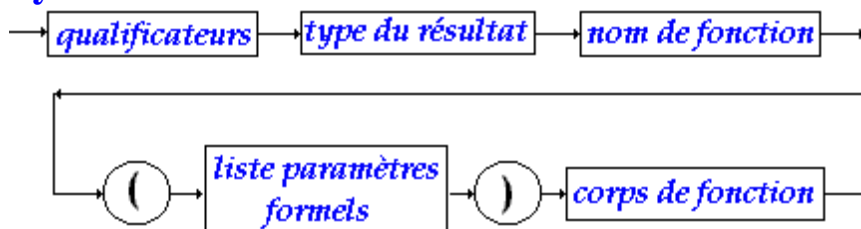
Donc par la suite dans ce document lorsque nous emploierons le mot méthode sans autre adjectif, il s'agira d'une **méthode de classe**, comme nos applications ne possèdent qu'une seule classe, nous pouvons assimiler ces méthodes aux fonctions de l'application et ainsi retrouver une *utilisation classique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

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

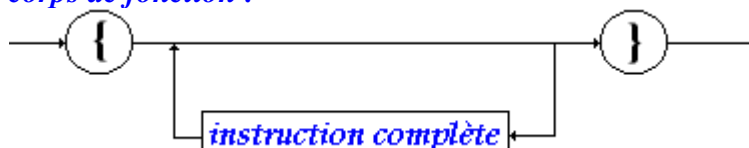
### Déclaration d'une méthode

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

#### Syntaxe :



#### corps de fonction :



Nous dénommons en-tête de fonction la partie suivante :

<qualificateurs><type du résultat><nom de fonction> (<liste paramètres formels>)

### Sémantique :

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

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

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

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

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

## Appel d'une méthode

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

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

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

**Appel de la méthode afficher**

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

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

**Appel de la méthode afficher**

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

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

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

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

### 3 - Transmission des paramètres

*Rappelons tout d'abord quelques principes de base :*

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

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

## Compatibilité des types des paramètres

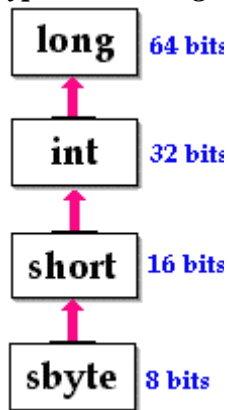
Resituons la compatibilité des types entier et réel en C#.

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

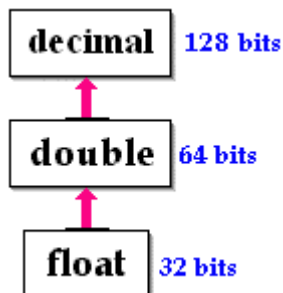
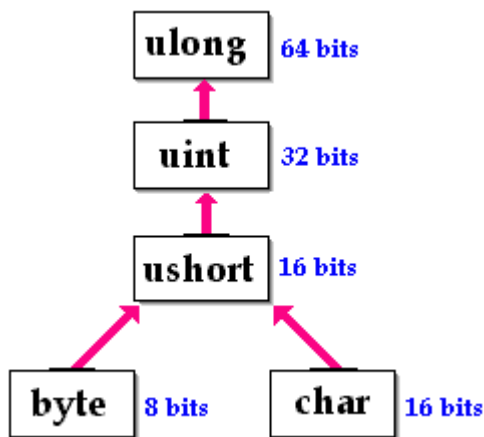
## Compatibilité ascendante des types de variables en C#

(conversion implicite sans transtypage obligatoire)

Les types entiers signés :



Les types entiers non signés :



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

```

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

```

### Appel de la méthode afficher

afficher(i,elt);

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

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

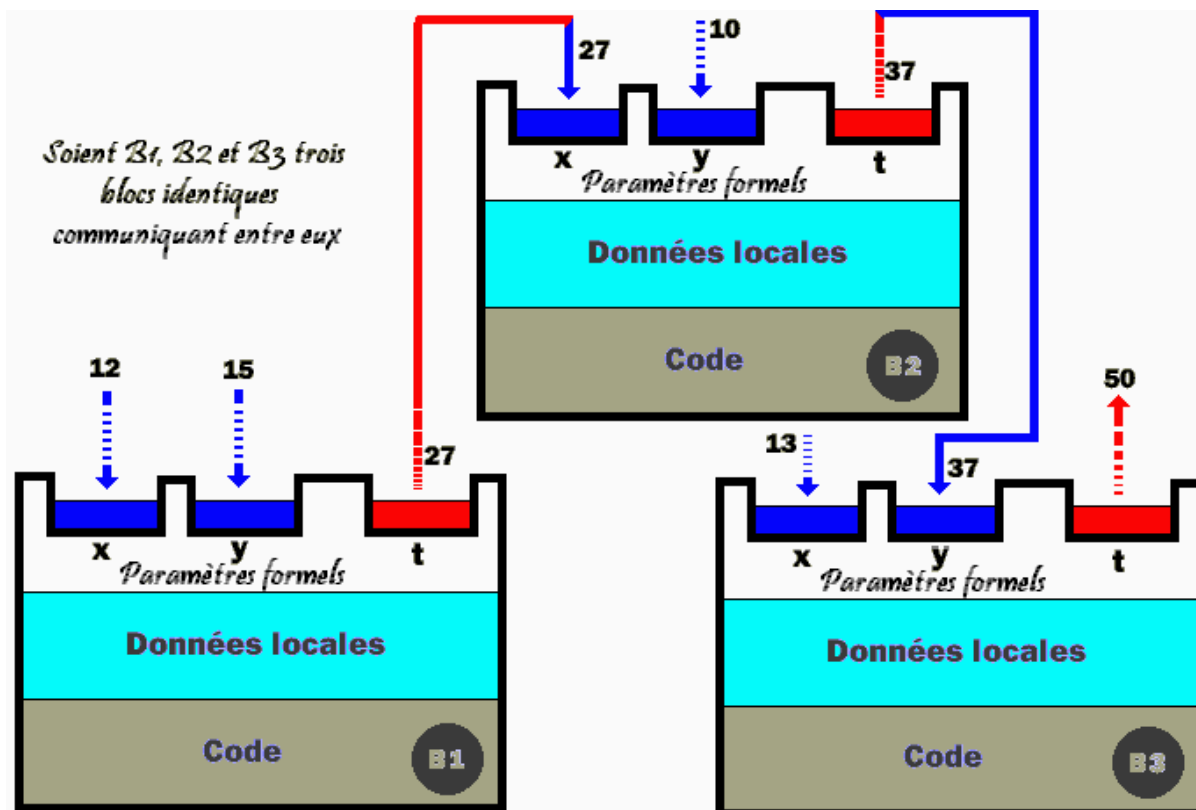
- Le paramètre effectif "elt" est associé au paramètre formel val. (sbyte = entier signé sur 8 bits et long = entier signé sur 64 bits)

Les trois modes principaux de transmission des paramètres sont :

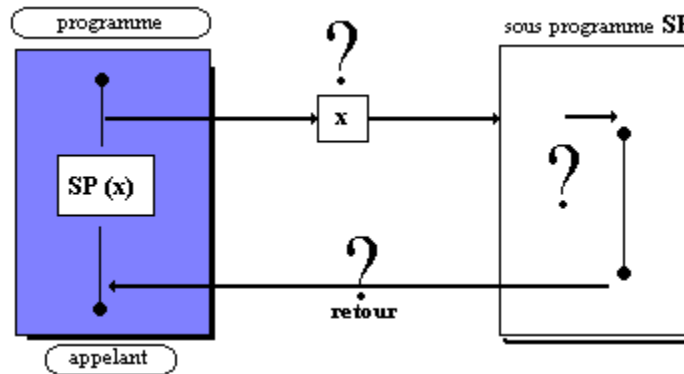


- passage par valeur
- passage par référence
- passage par résultat

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



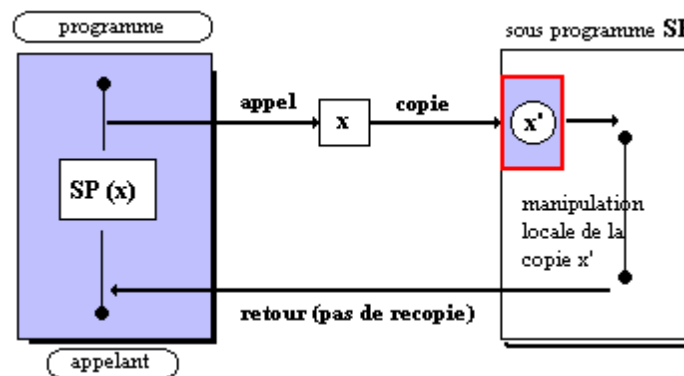
La question technique qui se pose en C# comme dans tout langage de programmation est de connaître le fonctionnement du passage des paramètres :



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

### 3.1 - Les paramètres C# passés par valeur

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



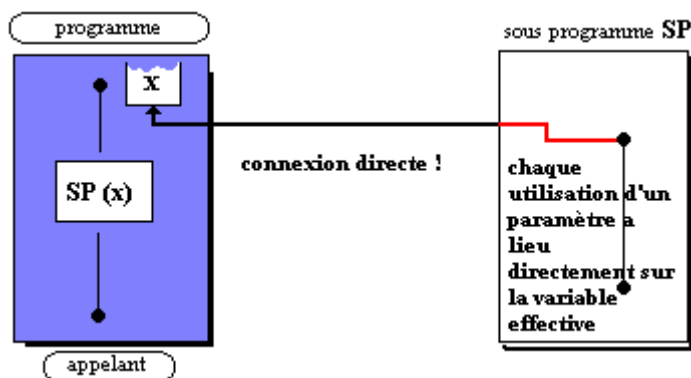
En C# tous les paramètres sont passés par défaut par valeur (lorsque le paramètre est un objet, c'est en fait **la référence de l'objet qui est passée par valeur**). Pour ce qui est de la vision algorithmique de C#, le passage par valeur permet à une variable d'être passée comme paramètre d'entrée.

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

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

### 3.2 - Les paramètres C# passés par référence

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



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

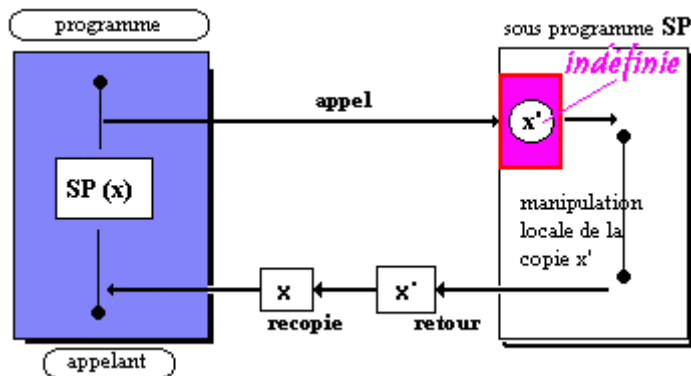
```
static int methode1(int a , ref char b) {
//.....
return a+b; }
```

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

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

### 3.3 - Les paramètres C# passés par résultat

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



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



**out :**

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

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

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

### Remarque :

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

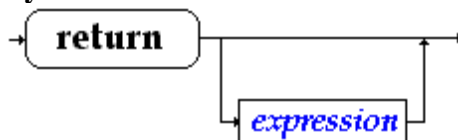
## 3.4 - Les retours de résultats de méthode type fonction

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

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

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

**Syntaxe :**



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

## 3.5 – Nombre de paramètres variables

C# autorise le passage d'un nombre de paramètres variables dans une méthode, il utilise une syntaxe spécifique pour cette opération.

### params

Le mot clef **params** permet de spécifier que le nombre des paramètres formels d'une méthode est variable. Ces paramètres sont obligatoirement déclarés dans un tableau.

### Contrainte et utilisation

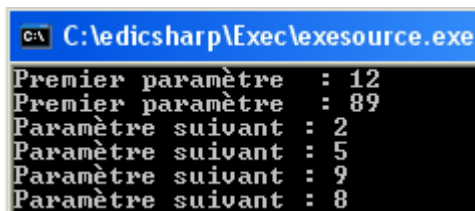
- Le mot clef **params** doit être le dernier de la liste des paramètres formels (aucune déclaration de paramètre après **params** n'est recevable).
- Un seul mot clef **params** est autorisé dans la liste des paramètres formels.
- Il peut y avoir des paramètres formels en nombre fixe, en sus des paramètres en nombre variable.
- Le passage ne peut s'effectuer que par **valeur** (pas de **ref**, pas de **out**)
- La liste des paramètres effectifs passés par **params** peut être vide.

Exemple :

```
class Exercice
{
    public static void paramsVariable( int nbr, params int[] listeParam )
    {
        Console.WriteLine("Premier paramètre : "+nbr);
        for (int i = 0 ; i < listeParam.Length; i++)
            Console.WriteLine("Paramètre suivant : "+listeParam[i]);
    }

    public static void Main( string[] x )
    {
        paramsVariable( 12 );
        paramsVariable( 89, 2, 5, 9, 8 );
        Console.ReadLine();
    }
}
```

Résultats de l'exécution :



```
C:\> C:\edics\harp\Exec\lexesource.exe
Premier paramètre : 12
Premier paramètre : 89
Paramètre suivant : 2
Paramètre suivant : 5
Paramètre suivant : 9
Paramètre suivant : 8
```

## 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 qualificatifs **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 avons 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

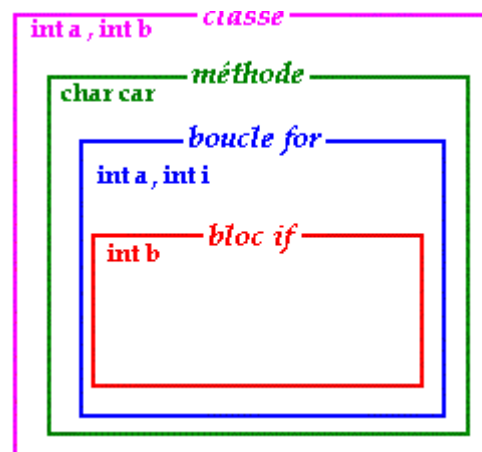
```
class ExempleVisible4 {
    static int a = 10, b = 2;

    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**.

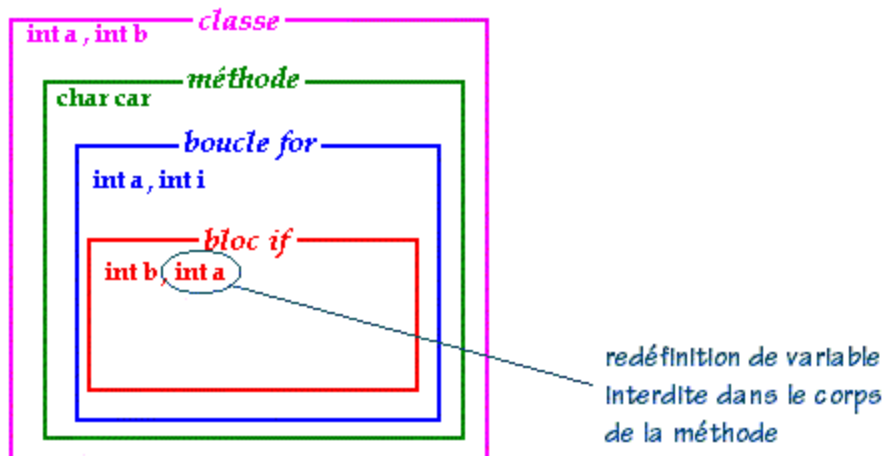
La variable de classe "b" définie dans **static int b = 2;** est masquée dans la méthode **f** dans le bloc imbriqué **if**.

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

Schéma de visibilité incorrect



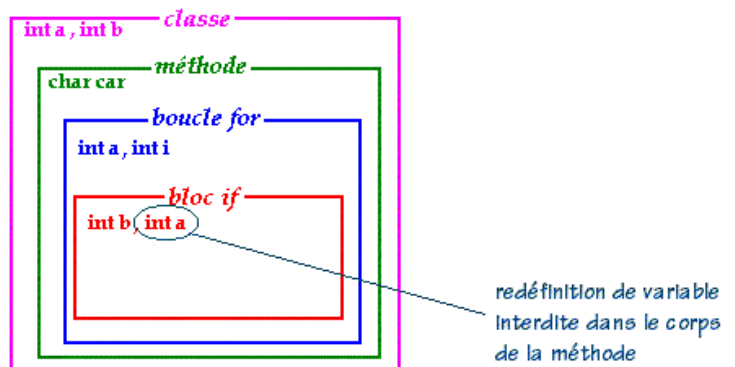
```

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;
      }
    }
}

```

Schéma de visibilité incorrect



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

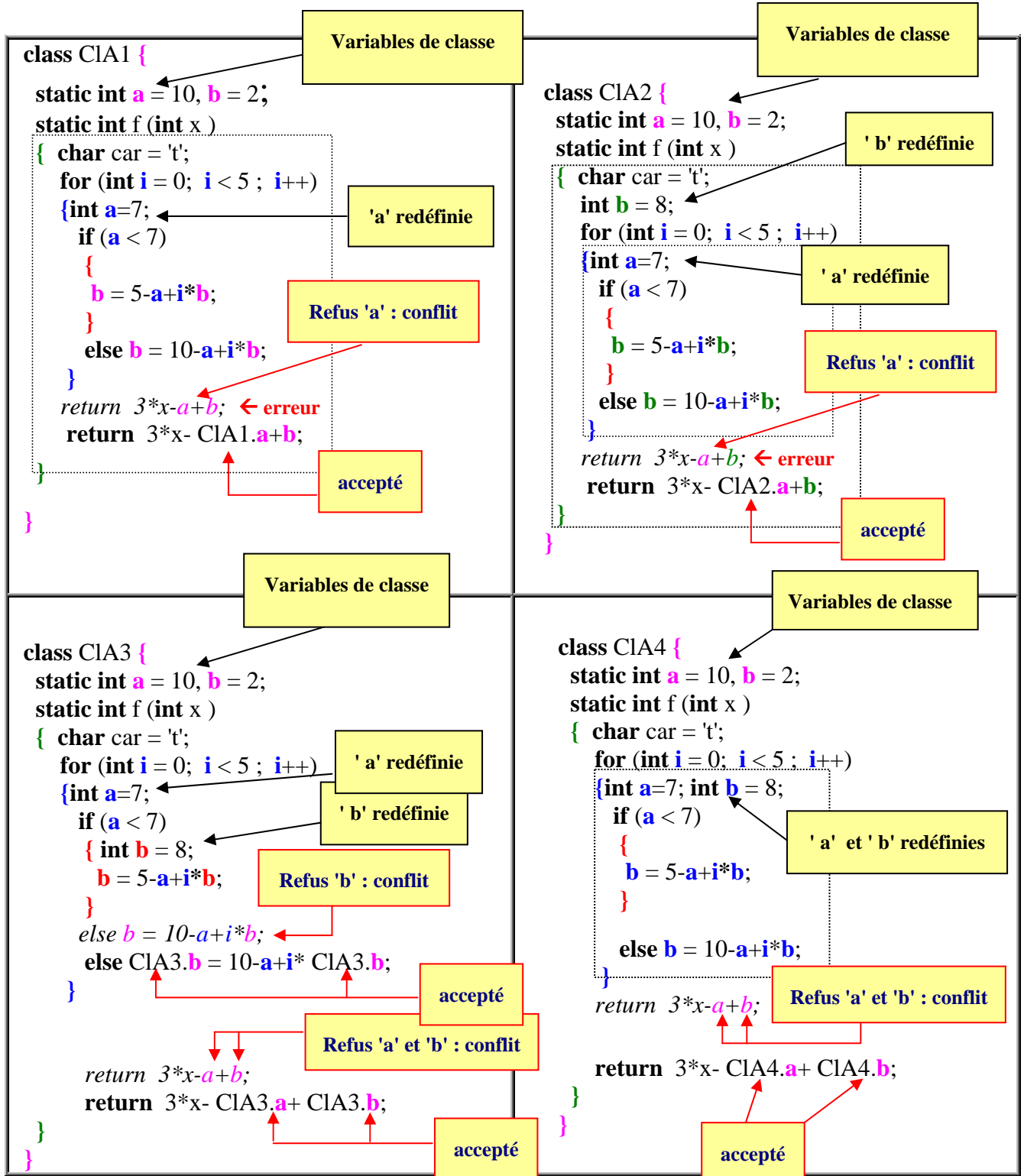
```
    return 3*x-a+b;  
}  
}
```

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

Remarquons que le principe de visibilité 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#

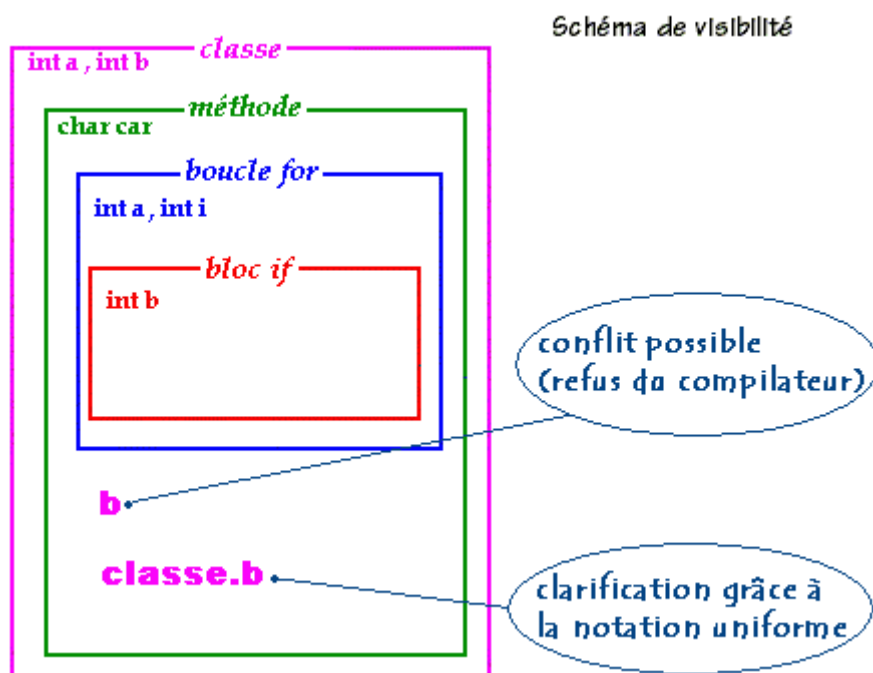
Le compilateur C# n'accepte pas que l'on utilise une **variable de classe** qui a été **redéfinie dans un bloc** interne à une méthode, sans la qualifier par la notation uniforme aussi appelée opération de résolution de portée. Les 4 exemples ci-dessous situent le discours :





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#
<pre> class CIA5 {   static int a = 10, b = 2;   static int f (int x )   { char car = 't';     for (int i = 0; i &lt; 5 ; i++)     {       if (a &lt; 7)       { int b = 8 , a = 7;         b = 5-a+i*b;       }       else         b = 10-a+i* b;     }     return 3*x- a+ b;   } } </pre>	<pre> class CIA5 {   static int a = 10, b = 2;   static int f (int x )   { char car = 't';     for (int i = 0; i &lt; 5 ; i++)     {       if (CIA5.a &lt; 7)       { int b = 8 , a = 7;         b = 5-a+i*b;       }       else         CIA5.b = 10-CIA5.a+i*CIA5.b;     }     return 3*x- CIA5.a+ CIA5.b;   } } </pre>

# Les chaînes de caractères string

## C#.net

### La classe string

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

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

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

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

### Accès à une chaîne string

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

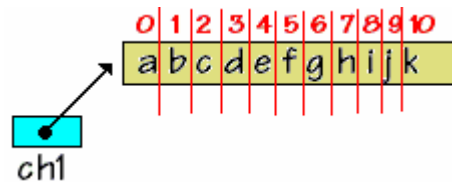
**On accède à un caractère de rang fixé d'une chaîne par l'opérateur [ ] :**

(la chaîne est lue comme un tableau de char)

Il est possible d'accéder en lecture seulement à chaque caractères d'une chaîne, mais qu'il est impossible de modifier un caractère directement dans une chaîne.

```
String ch1 = "abcdefghijk";
```

Représentation interne de l'objet ch1 de type **string** :



```
char car = ch1[4];
```

*// ici la variable car contient la lettre 'e'*

### Remarque

En fait l'opérateur [ ] est un indexeur de la classe **string** (cf. chapitre indexeurs en C#), et il est en lecture seule :

```
public char this [ int index ] { get ; }
```

Ce qui signifie au stade actuel de compréhension de C#, qu'il est possible d'accéder en lecture seulement à chaque caractère d'une chaîne, mais qu'il est impossible de modifier un caractère grâce à l'indexeur.

```
char car = ch1[7]; // l'indexeur renvoie le caractère 'h' dans la variable car.  
ch1[5] = car ; // Erreur de compilation : l'écriture dans l'indexeur est interdite !!  
ch1[5] = 'x' ; // Erreur de compilation : l'écriture dans l'indexeur est interdite !!
```

## Opérations de base sur une chaîne string

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

**Position d'une sous-chaîne à l'intérieur d'une chaîne donnée :**

**Méthode surchargée 6 fois :**

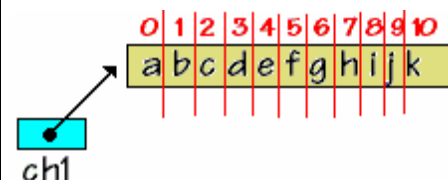
```
int IndexOf ( ...)
```

Ci-contre une utilisation de la surcharge :

```
int IndexOf ( string ssch) qui renvoie l'indice de la première occurrence du string ssch contenue dans la chaîne scannée.
```

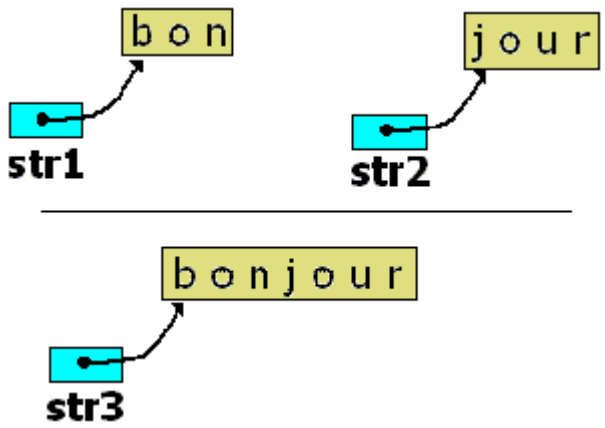
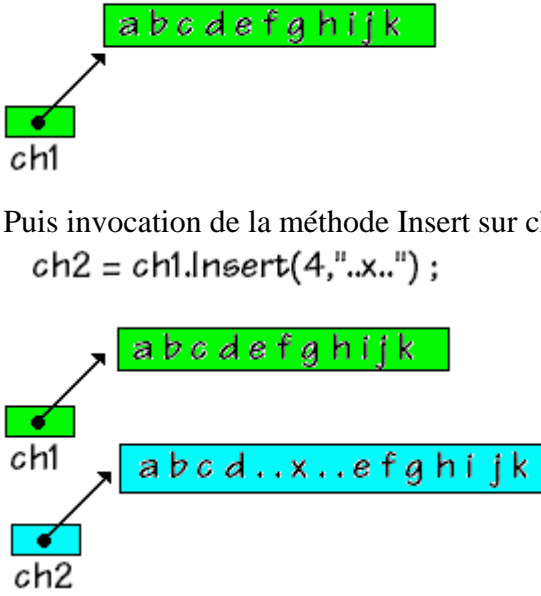
**Recherche de la position de ssch dans ch1 :**

```
String ch1 = " abcdef " , ssch="cde";
```



```
int rang ;  
rang = ch1.IndexOf ( ssch );
```

*// ici la variable rang vaut 2*

<p><b>Concaténation de deux chaînes</b></p> <p><b>Un opérateur ou une méthode</b></p> <p><b>Opérateur : + sur les chaînes</b></p> <p>ou</p> <p><b>Méthode static surchargée 8 fois :</b></p> <p><b>String</b> Concat(...)</p> <p>Les deux écritures ci-dessous sont donc équivalentes en C# :</p> <p>str3 = str1+str2 ⇔ str3 = str1.Concat(str2)</p>	<pre>String str1,str2,str3; str1="bon"; str2="jour"; str3=str1+str2;</pre> 
<p><b>Insertion d'une chaîne</b></p> <p><b>dans</b></p> <p><b>une autre chaîne</b></p> <p>Appel de la méthode <b>Insert</b> 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 (immutabilité).</p>	<p>Soit :</p> <pre>ch1 = "abcdefghijkl";</pre>  <p>Puis invocation de la méthode Insert sur ch1 :</p> <pre>ch2 = ch1.Insert(4,"..x..");</pre> <p><b>ch2 est une copie de ch1 dans laquelle on a inséré la sous-chaîne "..x..".</b></p>

**Attention :**

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

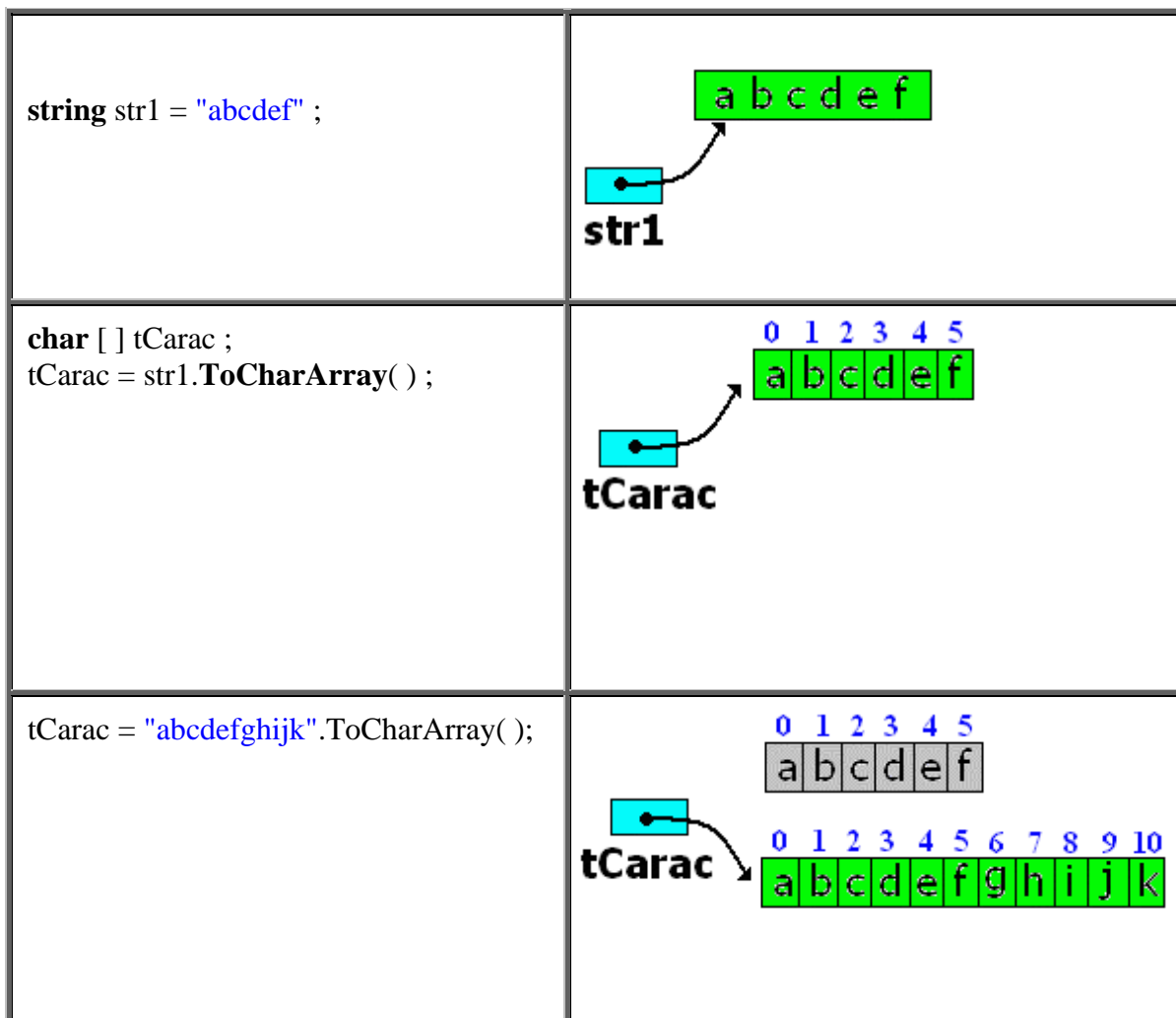
## Convertir une chaîne string en tableau de caractères

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 = "abcdefghijkl".ToCharArray( ) ;
```

Illustrons ces lignes par des schémas de références :



## 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;
```

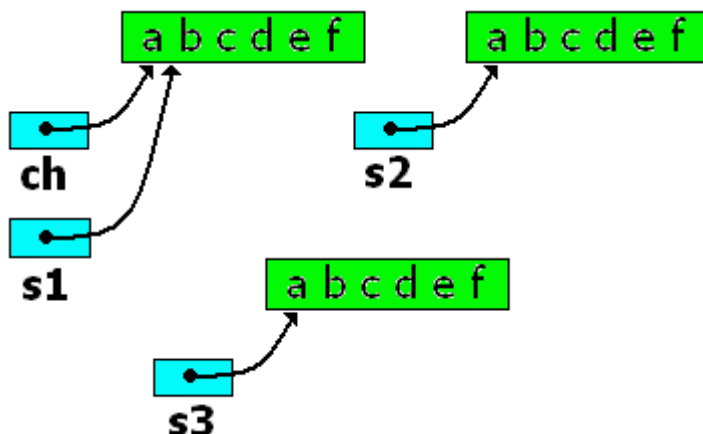
(`a == b`) renvoie **true** si la valeur de a est la même que la valeur de b ; sinon il renvoie **false**.

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

## 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 **Common Language Runtime**. 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 :
<pre>String s1 , s2 ="abc" ; char c = 'e' ; s1 = 'd' + c ; // types incompatibles</pre> <hr/> <pre>s1 = 'd' + 'e' ; // types incompatibles</pre>	<pre>String s1 , s2 ="abc" ; char c = 'e' ; s1 = "d" + Convert.ToString ( c ) ;</pre> <hr/> <pre>s1 = "d" + "e" ; s1 = "d" + 'e' ; s1 = 'd' + "e" ;</pre>

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

## C#.net

### Généralités sur les tableaux

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

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

Tableaux à une dimension.

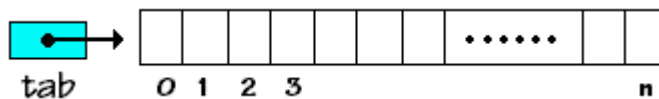
Tableaux à plusieurs dimensions (matrices,...) comme en Delphi.

Tableaux déchiquetés comme en Java.

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

### Tableau uni-dimensionnel

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



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

**Remarque :**

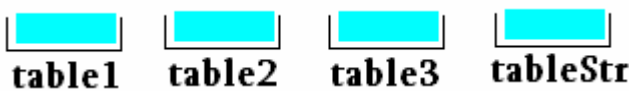
Les tableaux de C# sont des objets d'une classe dénommée **Array** qui est la classe de base d'implémentation des tableaux dans le CLR de .NET framework (localisation : **System.Array**) Cette classe n'est pas dérivable pour l'utilisateur : "**public abstract class Array : ICloneable, IList, ICollection,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

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

**Déclaration d'une variable de tableau, référence seule:**

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

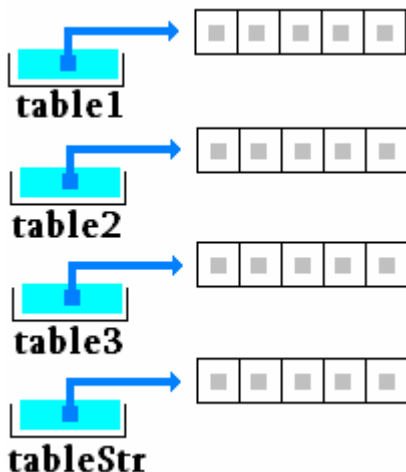
Voici une image de ce qui est présent en mémoire centrale après ces déclarations :



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

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

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



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

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

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

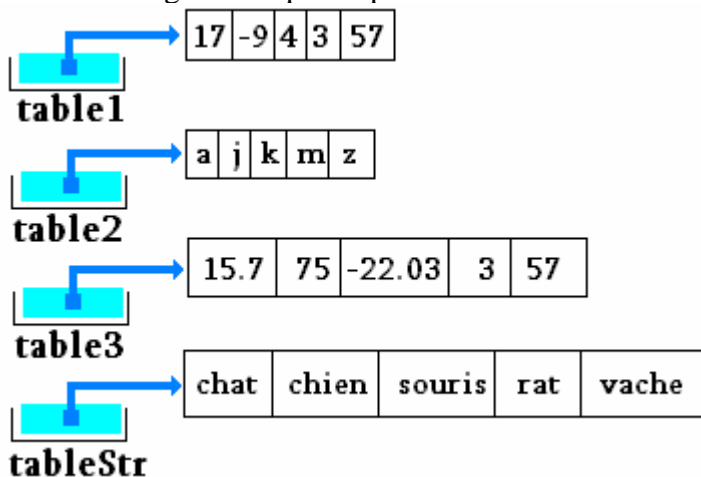
```
int [ ] table1;  
char [ ] table2;  
float [ ] table3;  
string [ ] tableStr;  
....  
table1 = new int [5];  
table2 = new char [12];  
table3 = new float [8];  
tableStr = new string [9];
```

### *Déclaration et initialisation avec définition implicite de taille :*

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

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

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



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

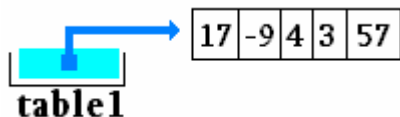
*Exemple :*

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

## Attention

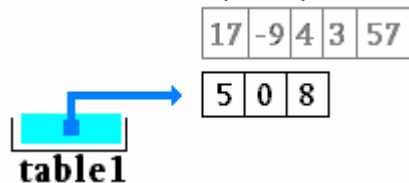
Il est possible de déclarer une référence de tableau, puis de l'initialiser après uniquement ainsi :

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



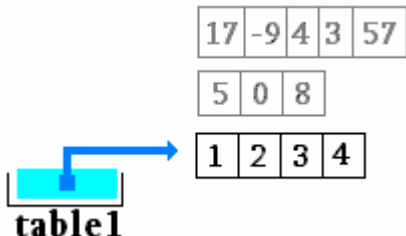
...

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



...

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



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

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

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

## Utiliser un tableau

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

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

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

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

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

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

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

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

### **Remarque :**

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

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

## Affectation de tableaux, copie de tableaux uni-dimensionnels

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

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

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

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

Instructions	Image en mémoire centrale
<pre>int []table1 = new int [9]; int []table2 = new int [6];</pre>	<p>table1</p> <p>table2</p>
<pre>for(int k = 0; k&lt;=8; k++)     table1(k) = k * 10 ;</pre>	<p>table1</p>
<pre>for(int k = 0; k&lt;=5; k++)     table2(k) = k ;</pre>	<p>table2</p>
<pre>table2 = table1;</pre>	<p>table1</p> <p>table2</p> <p>table1 et table2 maintenant se réfèrent à la même entité.</p>

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

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

En reprenant les tableaux précédents :

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



## Les matrices et les tableaux multi-dimensionnels

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

### Déclaration d'une matrice, référence seule:

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

### Déclaration d'une matrice avec définition explicite de taille :

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

### Exemple d'écriture de matrices de type int :

```
int [ , ] table1 = new int [2 , 3 ];// deux lignes de dimension 3 chacunes  
  
// dans une instruction d'affectation:  
table1[ 0 , 0 ] = -458;  
table1[ 2 , 5 ] = -3; <--- est une erreur de dépassement ! (valeur entre 0 et 1)  
table1[ 1 , 4 ] = 83; <--- est une erreur de dépassement ! (valeur entre 0 et 2)  
  
// dans une instruction de boucle:  
for (int i = 0 ; i<= 2; i++)  
    table1[1 , i ] = 3*i-1;  
  
// avec initialisation d'une variable dans la déclaration :  
int n ;  
int [ , ] table1 = new int [4 , n =3 ];// quatre lignes de dimension 3 chacunes
```

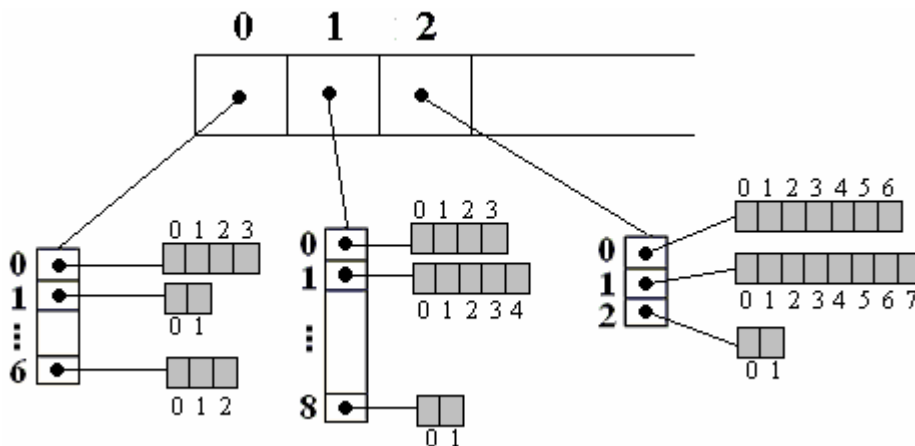
L'attribut **Length** en lecture seule, de la classe abstraite mère **Array**, contient en fait la **taille** d'un tableau en nombre total de cellules qui constituent le tableau (nous avons vu dans le cas uni-dimensionnel que cette valeur correspond à la taille de la dimension du tableau). Dans le cas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

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

## Tableaux déchiquetés ou en escalier

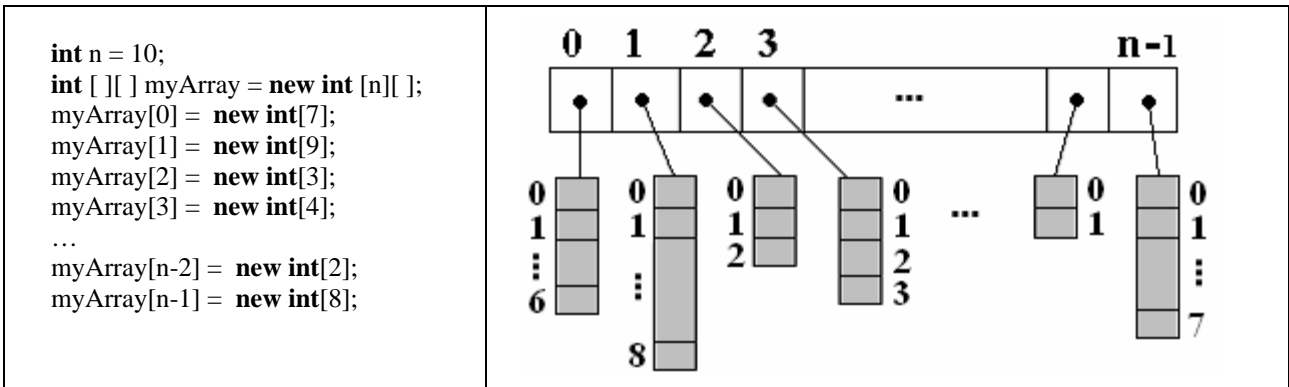
Leur structuration est strictement semblable à celle des tableaux Java, en fait en C# un tableau déchiqueté est composé de plusieurs tableaux unidimensionnels de taille variable. La déclaration s'effectue avec des opérateurs crochets [ ] [ ]... : autant de crochets que de dimensions. **C# autorise comme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

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



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

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



Ce tableau comporte  $7+9+3+4+\dots+2+8$  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;
```

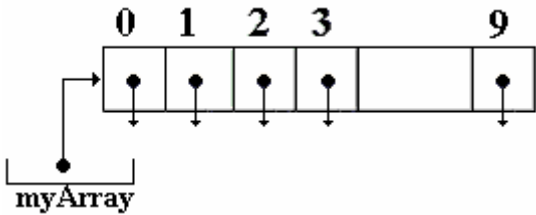
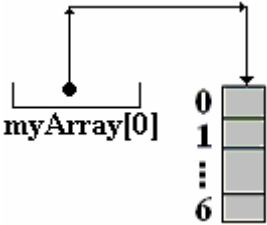
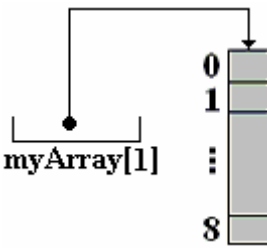
```
int [ ][ ] myArray = new int [n][ ];
```

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

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

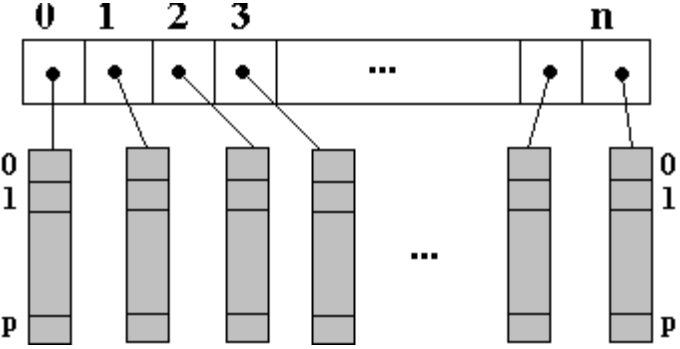
## Attention

Dans le cas d'un tableau décheté, le champ **Length** de la classe **Array**, contient la **taille** du sous-tableau unidimensionnel associé à la référence.

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

**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 :

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

### 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

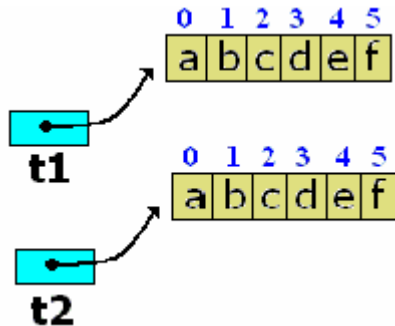
L'opérateur d'égalité `==` appliqué au tableau de n'importe quel type, détermine si deux objets spécifiés ont la même référence, il se comporte dans ce cas comme la méthode **Equals** de la classe **Object** qui ne teste que l'égalité de référence

```
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 crée deux tableaux de char t1 et t2, puis teste leur égalité avec l'opérateur `==` et la méthode **Equals** :

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



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

Après exécution on obtient :

```
t1<>t2  
t1 différent de t2
```

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

## Recopie de tableaux quelconques

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

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

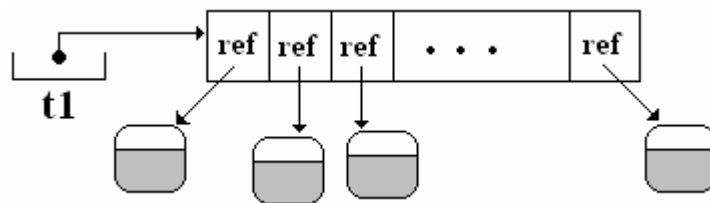
**public virtual object Clone()** : méthode qui renvoie une référence sur une nouvelle instance de

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

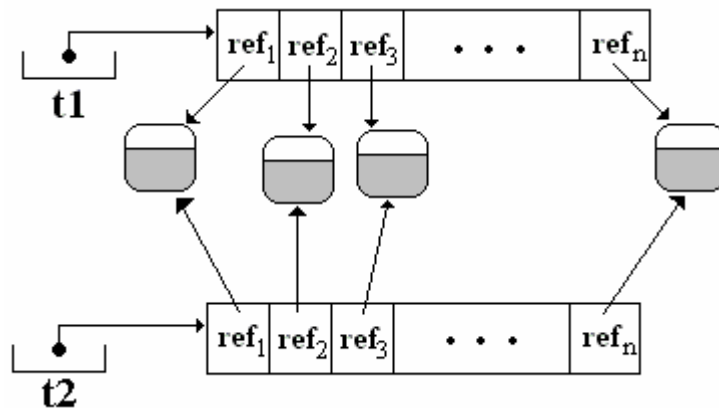
**public static void Copy ( Array t1 , Array t2, int long) :** méthode de classe qui copie dans un tableau t2 déjà existant et déjà instancié, long éléments du tableau t1 depuis son premier élément (si l'on veut une copie complète du tableau t1 dans t2, il suffit que long représente le nombre total d'éléments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



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



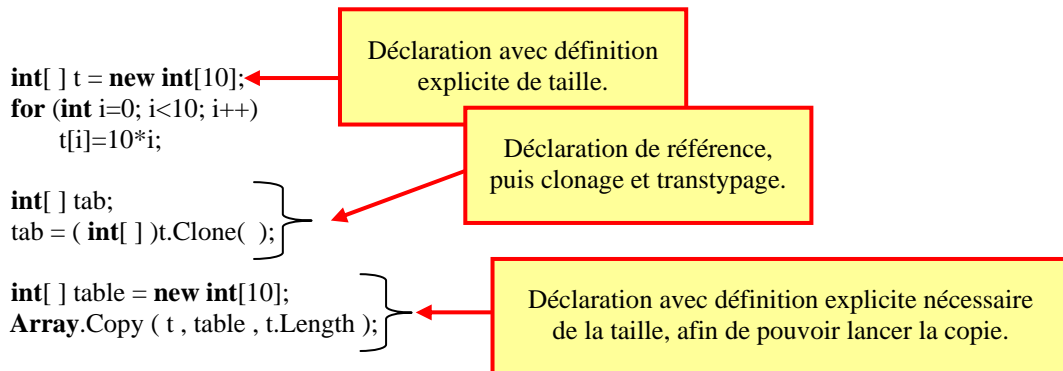
la recopie dans un autre tableau à travers les méthode Clone ou Copy ne recopie que les références, mais pas les objets pointés, voici un "clone" du tableau t1 de la figure précédente dans le tableau t2 :



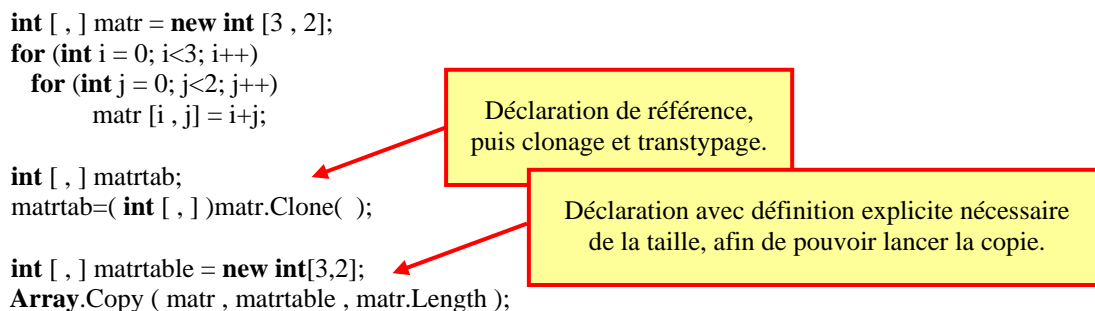
Si l'on veut que le clonage (la recopie) soit plus complète et comprenne aussi les objets pointés, il suffit de construire une telle méthode car malheureusement la classe abstraite **Array** n'est pas implantable par l'utilisateur mais seulement par le compilateur et nous ne pouvons pas redéfinir la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

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

*//-- tableau à une dimension :*



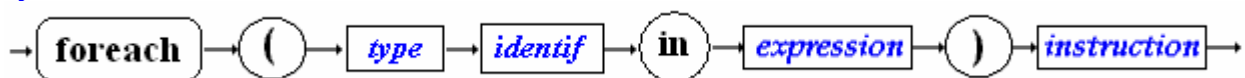
*//-- tableau à deux dimensions :*



## Parcours itératifs de tableaux - foreach...in

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

*Syntaxe*



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

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

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

### foreach...in dans un tableau uni-dimensionnel

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

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

<b>int [ ] table = new int [ 10 ]; ... Remplissage du tableau</b>	
<b>for</b> (int i=0; i<10; i++) System.Console.WriteLine ( table[i] );	<b>foreach</b> ( 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*) :

<b>int [ , ] table = new int [ 3 , 2 ]; ... Remplissage de la matrice</b>	
<b>for</b> (int i=0; i<3; i++) <b>for</b> (int j=0; j<2; i++) System.Console.WriteLine ( table[ i , j ] );	<b>foreach</b> ( 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

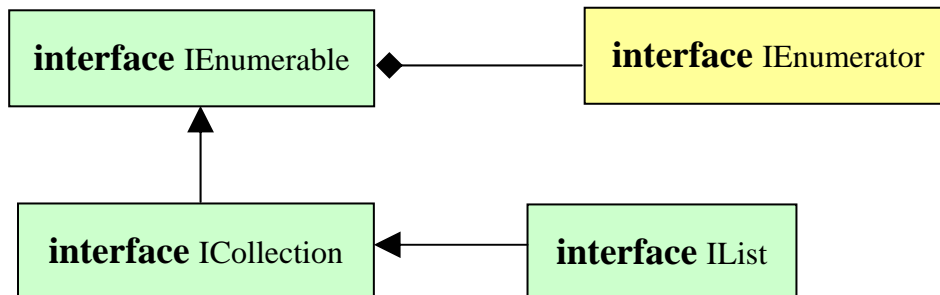
## C#.net

### Généralités sur les collections

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

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

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



#### IEnumerable :

contient une seule méthode qui renvoie un énumérateur (objet de type IEnumerator) qui peut itérer sur les éléments 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);**

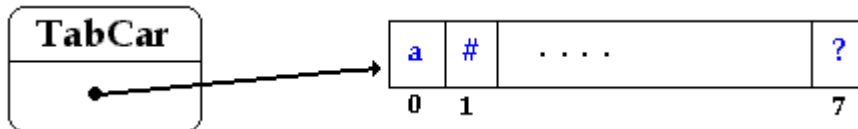
Supprime l'élément de **IList** dont le *rang* est spécifié.

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

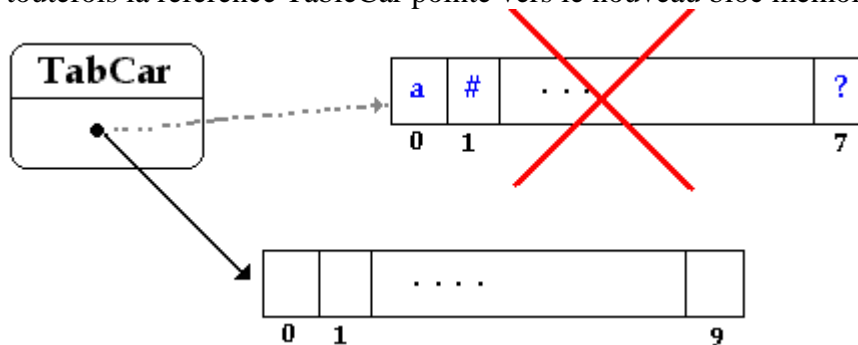
Un tableau Array à une dimension, lorsque sa taille a été fixée soit par une définition explicite, soit par une définition implicite, **ne peut plus changer de taille**, c'est donc un objet de taille statique.

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



*Si l'on rajoute l'instruction suivante aux précédentes*

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



Ce qui nous donne après exécution de la liste des instructions ci-dessous, un tableau TabCar ne contenant plus rien :

```
char [ ] TableCar ;  
TableCar = new char[8];  
TableCar[0] = 'a';  
TableCar[1] = '#';  
...  
TableCar[7] = '?';  
TableCar = new char[10];
```

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

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

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

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

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

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

**Schéma interne d'un ArrayList :**

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

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

```

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

```

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

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

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

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

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

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

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

IList  
ICollection  
IEnumerable  
ICloneable

## ArrayList

Class

### Properties

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

Classe ArrayList

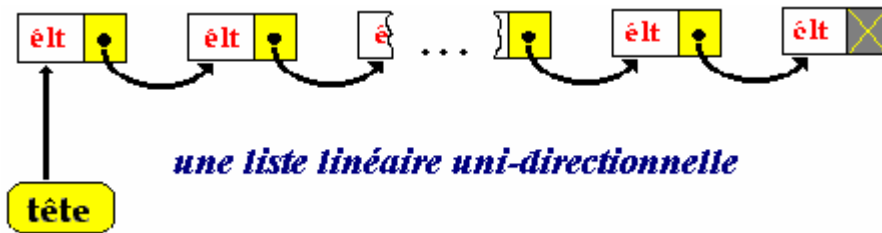
### Methods

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

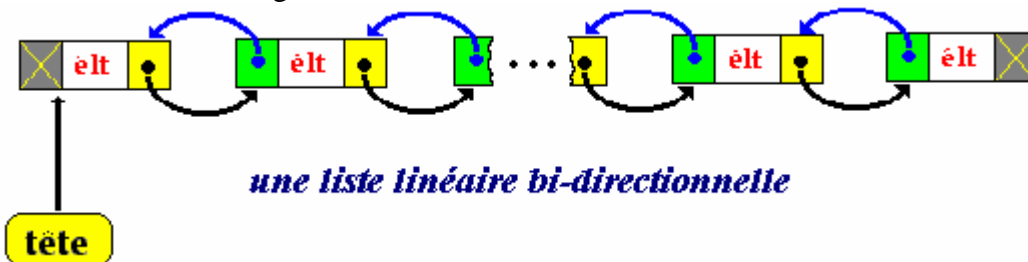
## Les listes chaînées : classe ArrayList

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

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



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



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

## Table de hachage - Hashcode

Une méthode de hachage calcule la place d'un élément dans une structure de données de collection à partir de sa clef. La recherche d'un élément grâce à sa clef est une recherche "associative". Nous nommons Dictionnaire une telle collection.

**TAD** Dictionnaire

**Utilise :** EnsClé, EnsElement

**Champs :**  $(a_1, \dots, a_n)$  suite finie dans EnsElement

**Opérations :**

Clef : EnsElement  $\rightarrow$  EnsClé

Ajouter : EnsElement x Collection  $\rightarrow$  Collection

Supprimer : EnsClé x Collection  $\rightarrow$  Collection

Rechercher : EnsClé x Collection  $\rightarrow$  EnsElement

**Fin TAD** Dictionnaire

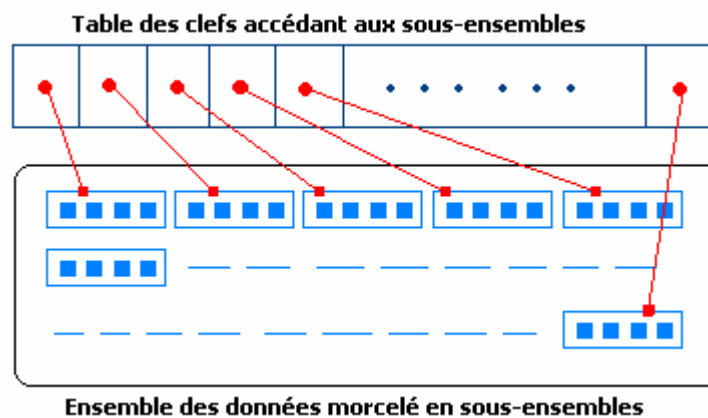
Comme la place d'un élément dans la collection Dictionnaire est obtenue à partir d'une fonction de calcul sur la clef de cet élément (fonction dite de hachage ou hashcode), l'idéal serait que cette fonction de hachage soit bijective afin d'avoir une place unique par chaque clef d'élément de la collection.

Toutefois dans la réalité le nombre des éléments du Dictionnaire est très grand et si en mémoire externe (disque dur) il n'y a pas de problème de place pour ranger ces éléments, il n'en est pas de même en mémoire centrale où la place est plus restreinte.

Par exemple, ranger des clients selon leur clef = nom + prénom, soit sur un maximum de 30 lettres, ce qui nous donne  $30^{26}$  possibilités de clefs différentes (de l'ordre de  $10^{38}$  clefs), il n'est donc pas possible d'utiliser une structure de donnée statique pour réserver une place en mémoire centrale pour chaque clef. Nous allons définir une structure classique pour stocker ces données associatives.

## Table de hachage

On décide de découper l'ensemble des éléments en  $N$  sous-ensembles (on fixe par exemple  $N=1000$ ) et de définir une structure de donnée statique de type table dont chaque cellule accède à un sous-ensemble de la collection, cette table se dénomme **table de hachage**.

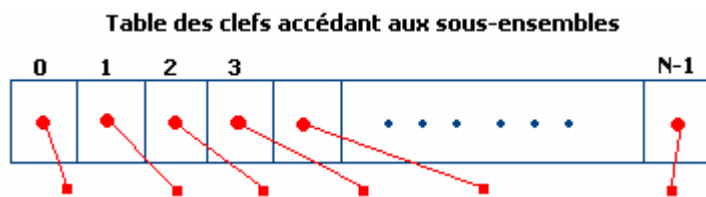


On construit une fonction de hachage que nous nommons Hashcode et qui à chaque clef d'un élément de



EnsElement fournit un entier compris dans l'intervalle [0 , N-1] :

Hascode : EnsClé  $\rightarrow$  [0 , N-1]



Pour toute clef  $k \in \text{EnsClé}$  , l'entier  $\text{Hashcode}(k)$  est appelé la valeur de hachage ou code de hachage primaire de cette clef.

Ce qui donne un TAD Dictionnaire enrichi :

**TAD** Dictionnaire

**Utilise** : EnsClé, EnsElement

**Champs** :  $(a_1, \dots, a_n)$  suite finie dans EnsElement

**Opérations** :

Clef : EnsElement  $\rightarrow$  EnsClé

Hashcode : EnsClé  $\rightarrow$  [0 , N-1]

Ajouter : EnsElement x Collection  $\rightarrow$  Collection

Supprimer : EnsClé x Collection  $\rightarrow$  Collection

Rechercher : EnsClé x Collection  $\rightarrow$  EnsElement

**Fin TAD** Dictionnaire

## Collision primaire

La fonction **Hashcode ( Clef ( element ) )** n'est pas bijective et même pas injective, puisque plusieurs éléments distincts peuvent avoir le même code de hachage primaire.

On dit qu'il y a collision entre deux éléments  $\text{elt}_1$  et  $\text{elt}_2$  lorsqu'ils sont dans le même compartiment (ils appartiennent au même sous-ensemble) :

**Hashcode ( Clef ( elt\_1 ) ) = Hashcode ( Clef ( elt\_2 ) )**

et

**Clef ( elt\_1 )  $\neq$  Clef ( elt\_2 )**

Ce qui veut dire que la clef d'un élément est insuffisante pour accéder aux informations sur l'élément, car la collision fait que plusieurs éléments sont situés dans la même zone associée au hashcode de cette clef, il est nécessaire de résoudre le problème posé par cette collision.

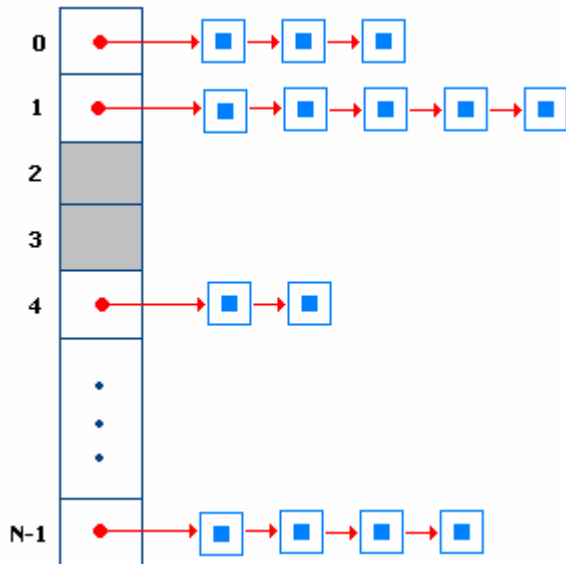
## Résolution des collisions par chaînage séparé

Il existe plusieurs stratégies d'implantation d'une telle résolution, elles sont toutes fondées sur le chaînage, nous donnons ici la stratégie de hachage ouvert par chaînage séparé qui est la plus performante dans le cas de la gestion dynamique de la mémoire.

Les éléments en collision (dans le même sous-ensemble de hachage) sont rangés sous forme de liste chaînée.



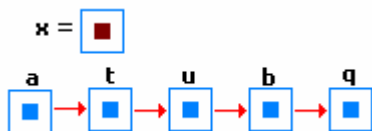
Chaque cellule de la table de hachage contient uniquement le pointeur sur la tête de liste :



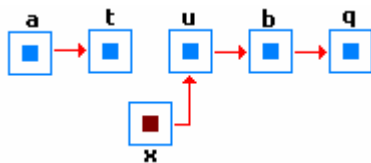
Les algorithmes ajouter, supprimer et rechercher un **élément x**, sont semblables à ceux qui permettent de manipuler les listes chaînées et sont proportionnels au pire, au nombre d'éléments en collision. Ils sont précédés par un accès fonctionnel direct dans la table à l'aide de **Hashcode ( Clef ( x ) )**.

## Ajouter

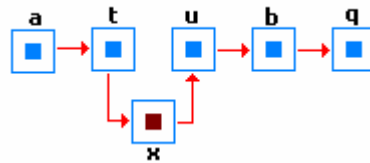
Soit à insérer l'élément **x** entre l'élément **t** et l'élément **u** dans le compartiment de collision contenant les éléments **a, t, u, b** et **q**.



Etape n°1 :



Etape n°2 :

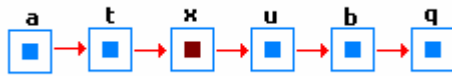


Compartiment mis à jour :

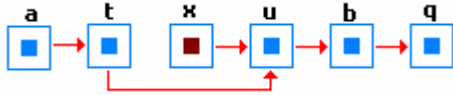


## Supprimer

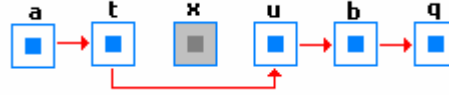
Soit à supprimer l'élément **x** entre l'élément **t** et l'élément **u** dans le compartiment de collision contenant les éléments **a**, **t**, **u**, **b** et **q**.



Etape n°1 :



Etape n°2 :

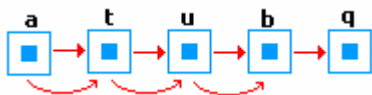


Compartiment mis à jour :



## Rechercher

Soit à rechercher l'élément **b** dans le compartiment de collision contenant les éléments **a**, **t**, **u**, **b** et **q**, par une recherche linéaire classique.



## Exemple de fonction de hachage

Soit le rangement des clients selon leur clef = nom + prénom, d'un maximum de 30 lettres dans une table de hachage de 10 000 cellules ( $N=10000$ ).

La fonction de hachage doit alors renvoyer un nombre dans l'intervalle  $[0, 9999]$ , la clef est une chaîne constitué de caractères accessibles par leur indice  $clef_k$ . Nous calculons la somme des valeurs du code Ascii de chacun des caractères composant la clef et nous prenons le reste de ce nombre modulo  $N$ , ce qui nous donne un nombre compris entre  $[0, N-1]$  :

$$\text{Hashcode}( \text{clef} ) = ( \sum \text{Ascii}( \text{clef}_k ) ) \text{ modulo } N$$

### Remarque

Des combinaisons différentes ou identiques de lettres peuvent donner le même hashcode. **ravale** et **avaler** ont le même hashcode, car ces deux chaînes sont constituées des mêmes lettres mais dans un autre ordre.  
Il faut que  $N$  soit assez grand pour ne pas avoir des listes trop grandes et que les valeurs obtenus à partir des clefs soit le plus possible équiréparties.

## Classe Hashtable dans NetFramework (dans System.Collections)

**Hashtable**  
Class

**Propriétés**

- comparer { get; set; } : IComparer
- Count { get; } : int
- EqualityComparer { get; } : IEqualityComparer
- hcp { get; set; } : IHashCodeProvider
- IsFixedSize { get; } : bool
- IsReadOnly { get; } : bool
- IsSynchronized { get; } : bool
- Item[object key] { get; set; } : object
- Keys { get; } : ICollection
- SyncRoot { get; } : object
- Values { get; } : ICollection

**Méthodes**

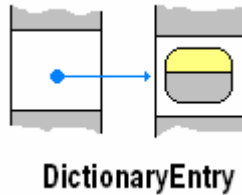
- Add(object key, object value) : void
- Clear() : void
- Clone() : object
- Contains(object key) : bool
- ContainsKey(object key) : bool
- ContainsValue(object value) : bool
- CopyTo(Array array, int arrayIndex) : void
- GetEnumerator() : IDictionaryEnumerator
- GetHash(object key) : int
- GetObjectData(SerializationInfo info, StreamingContext context) : void
- Hashtable()
- Hashtable(IDictionary d)
- Hashtable(IDictionary d, float loadFactor)
- Hashtable(IDictionary d, float loadFactor, IEqualityComparer equalityComparer)
- Hashtable(IDictionary d, float loadFactor, IHashCodeProvider hcp, IComparer comparer)
- Hashtable(IDictionary d, IEqualityComparer equalityComparer)
- Hashtable(IDictionary d, IHashCodeProvider hcp, IComparer comparer)
- Hashtable(IEqualityComparer equalityComparer)
- Hashtable(IHashCodeProvider hcp, IComparer comparer)
- Hashtable(int capacity)
- Hashtable(int capacity, float loadFactor)
- Hashtable(int capacity, float loadFactor, IEqualityComparer equalityComparer)
- Hashtable(int capacity, float loadFactor, IHashCodeProvider hcp, IComparer comparer)
- Hashtable(int capacity, IEqualityComparer equalityComparer)
- Hashtable(int capacity, IHashCodeProvider hcp, IComparer comparer)
- Hashtable(SerializationInfo info, StreamingContext context)
- KeyEquals(object item, object key) : bool
- OnDeserialization(object sender) : void
- Remove(object key) : void
- Synchronized(Hashtable table) : Hashtable

Représente une collection de paires clé/valeur qui sont organisées en fonction du code de hachage de la clé, cette classe assure automatiquement la gestion des collisions.

**Informations Microsoft :**

Un objet **Hashtable** se compose de compartiments contenant les éléments de la collection. Un compartiment est un sous-groupe virtuel d'éléments dans le type **Hashtable**, ce qui rend les opérations de recherche et d'extraction plus simples et plus rapides que dans la plupart des collections. Chaque compartiment est associé à un code de hachage, généré à l'aide d'une fonction de hachage et reposant sur la clé de l'élément.

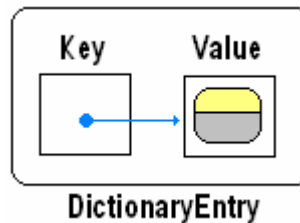
Chaque élément est une paire clé/valeur stockée dans un objet **DictionaryEntry**. Une clé ne peut pas être référence Null (**Nothing** en Visual Basic), contrairement à une valeur.



**public struct DictionaryEntry**

Définit une paire clé/valeur du dictionnaire qui peut être définie ou récupérée.

The screenshot shows the Visual Studio interface for the DictionaryEntry struct. It lists the following properties: Key { get; set; } : object and Value { get; set; } : object. It also lists a constructor: DictionaryEntry(object key, object value).



**Membres importants de Hashtable**

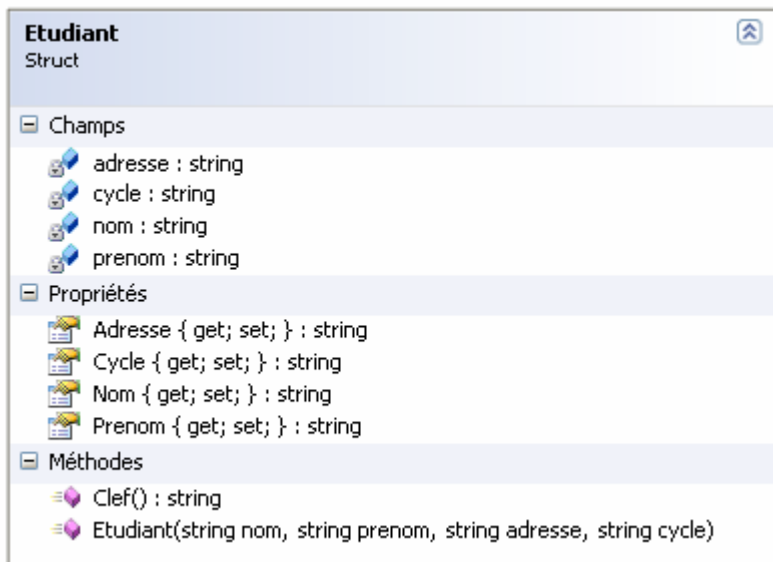
Signature	Description
<p><b>public virtual ICollection Keys { get; }</b> Obtient ICollection contenant les clés de Hashtable.</p> <p><b>public virtual ICollection Values { get; }</b> Obtient ICollection contenant les valeurs de Hashtable.</p>	<p>La table de hachage</p> <p>The diagram shows a vertical table with two rows. The left column is labeled "Keys ICollection" and the right column is labeled "Values ICollection". Each row contains a blue dot (key) and a colored oval (value). The top row has a cyan value, and the bottom row has a yellow value. Blue arrows point from each key to its corresponding value.</p>
<p><b>public virtual object this [ object key ]</b> Obtient ou définit la valeur associée à la clé spécifiée.</p>	<p><b>Rechercher</b> un élément par sa clef unique.</p>
<p><b>public virtual void Add ( object key, object value )</b> Ajoute un élément avec la clé et la valeur spécifiées dans Hashtable.</p>	<p><b>Ajouter</b> un élément et sa clef unique.</p>

**public virtual void Remove ( object key )**  
*Supprime de Hashtable l'élément ayant la clé spécifiée.*

**Supprimer** un élément par sa clef unique.

## Exercice :

Construire une classe **ClasseHash** dérivant de **Hashtable** permettant de stocker des informations sur des objets de type **Etudiant** :

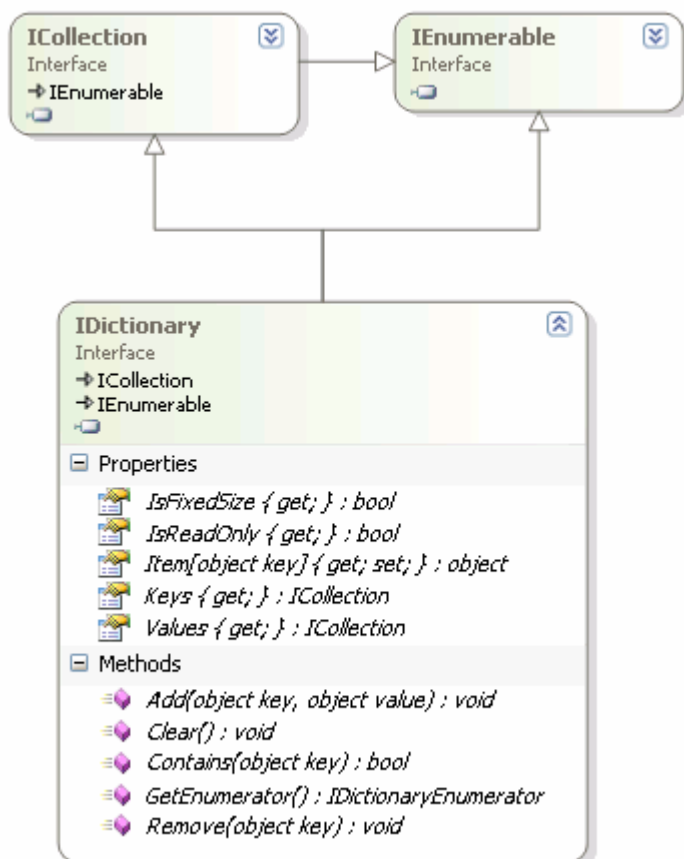


## Liste à clefs triées : classe SortedList

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

### Les interfaces implémentées par SortedList

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



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

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

### La classe SortedList :

The screenshot shows the SortedList class in Visual Studio. It is a class that inherits from IDictionary, ICollection, IEnumerable, and ICloneable. The class has several properties and methods listed below.

**SortedList**  
Class

**Properties**

- Capacity { get; set; } : int
- Count { get; } : int
- IsFixedSize { get; } : bool
- IsReadOnly { get; } : bool
- IsSynchronized { get; } : bool
- Item[object key] { get; set; } : object
- Keys { get; } : ICollection
- SyncRoot { get; } : object
- Values { get; } : ICollection

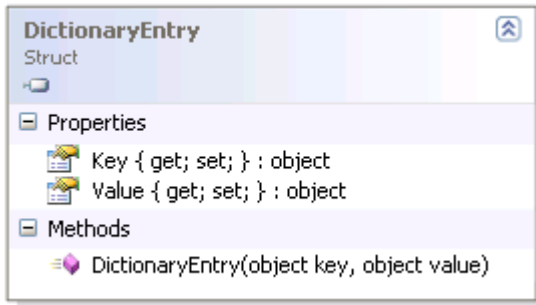
**Methods**

- Add(object key, object value) : void
- Clear() : void
- Clone() : object
- Contains(object key) : bool
- ContainsKey(object key) : bool
- ContainsValue(object value) : bool
- CopyTo(Array array, int arrayIndex) : void
- GetByIndex(int index) : object
- GetEnumerator() : IDictionaryEnumerator
- GetKey(int index) : object
- GetKeyList() : IList
- GetValueList() : IList
- IndexOfKey(object key) : int
- IndexOFValue(object value) : int
- Remove(object key) : void
- RemoveAt(int index) : void
- SetByIndex(int index, object value) : void
- SortedList()
- SortedList(IComparer comparer)
- SortedList(IComparer comparer, int capacity)
- SortedList(IDictionary d)
- SortedList(IDictionary d, IComparer comparer)
- SortedList(int initialCapacity)
- Synchronized(SortedList list) : SortedList
- TrimToSize() : void

### Structure utilisée par SortedList

Structure	Description
DictionaryEntry	Définit un unique couple (clef, valeur) qui peut être stocké ou retrouvé. <b>public struct</b> DictionaryEntry





### Schéma interne d'un SortedList :

( un SortedList = 2 tables **Keys** et **Values** liées entre elles implémentant un IDictionnaire )

Où **n** est le nombre d'éléments présent dans le **SortedList** (  $n = \text{this.Count}$  ).

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

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

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

**DictionaryEntry**

**DictionaryEntry**

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

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

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

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

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

### 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) :

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

```
Liste.GetByIndex( 2 ) : Murielle
Liste [ 45 ] : Murielle
Liste.GetValueList( ) [ 2 ] : Murielle
```

Les trois boucles affichent dans l'ordre :

```
Luc
José
Murielle
Jean
Claudie
```

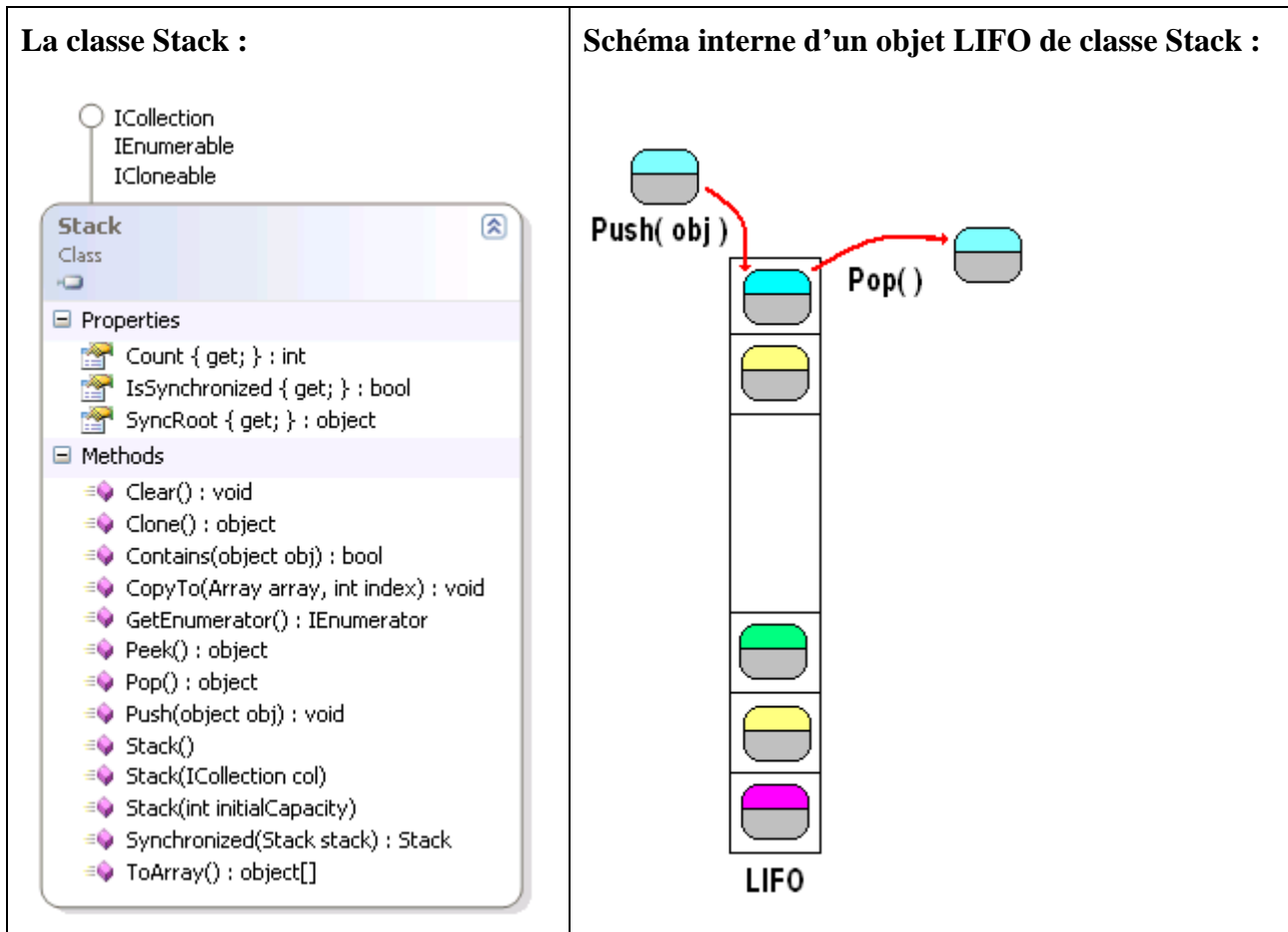
## Piles Lifo, files Fifo : classes Stack et Queue

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

```
System.Collections.Stack
System.Collections.Queue
```

### Les interfaces implémentées par Stack

Interface	Description
<b>ICollection</b>	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
<b>ICloneable</b>	Contrat d'une seule méthode pour cloner (recopier entièrement) un objet : <b>object clone( )</b>
<b>IEnumerable</b>	Contrat pour l'énumération d'une collection.

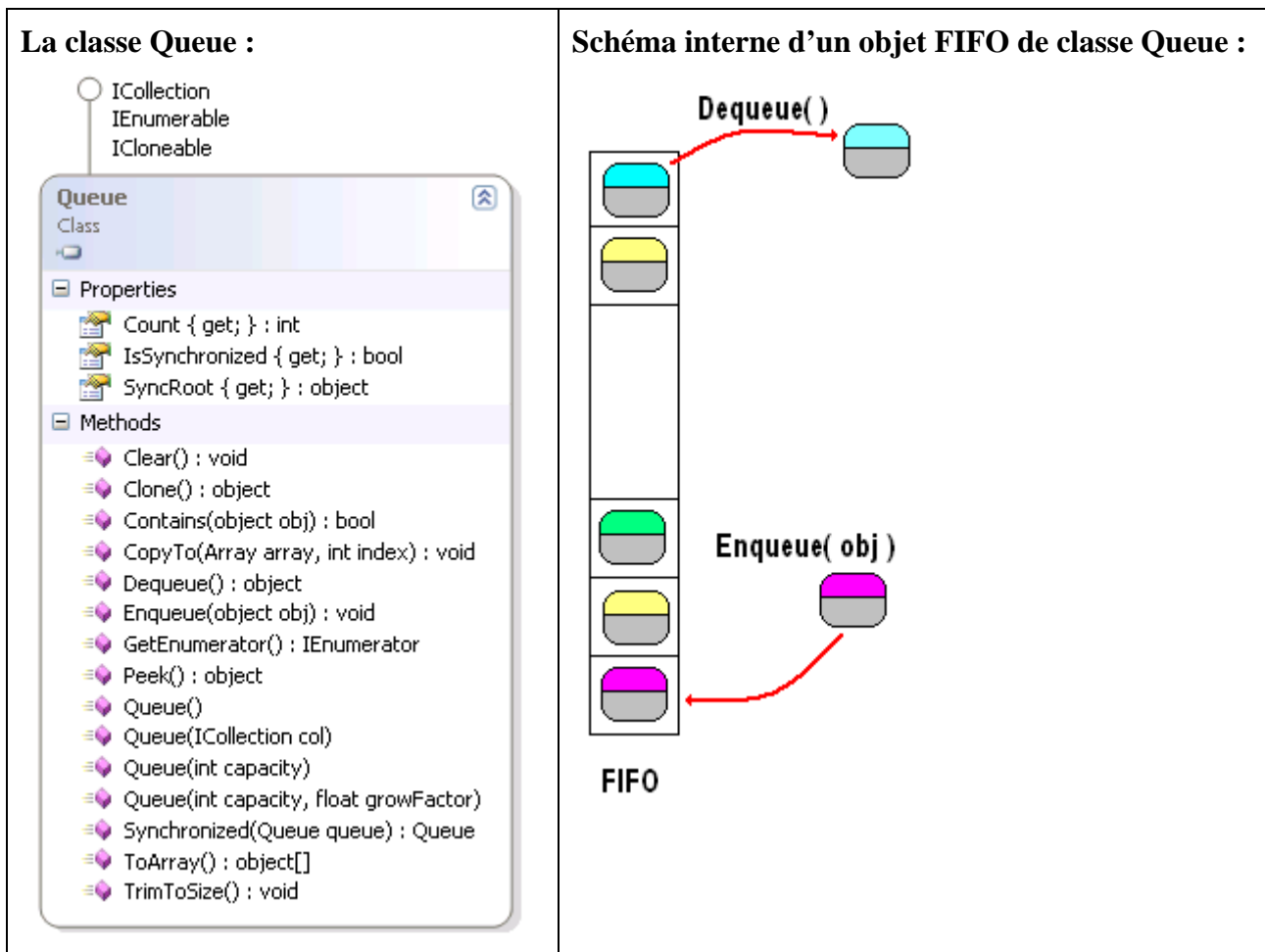


La classe "**public class Stack : ICollection, IEnumerable, ICloneable**" représente une pile Lifo :

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

### Les interfaces implémentées par Queue

Interface	Description
<b>ICollection</b>	Contrat pour la taille, l'énumération et la synchronisation d'une collection.
<b>ICloneable</b>	Contrat d'une seule méthode pour cloner (recopier entièrement) un objet : <b>object clone( )</b>
<b>IEnumerable</b>	Contrat pour l'énumération d'une collection.



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

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

### *Exemple d'utilisation d'une Lifo de type Stack*

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

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

<pre> class Lifo : Stack {     public virtual void getArray(string[] t) {         foreach(string s in t)             this.Push (s);     } } </pre>	<pre> 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) ;     } } </pre>
--	--

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

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

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

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

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

Voici le code de la nouvelle méthode ToArray :

<pre> 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 ;     } } </pre>	<pre> 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) ;     } } </pre>
--	--

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

fin  
fff  
eee  
ddd  
ccc  
bbb  
aaa

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


### Exemple d'utilisation d'une Fifo de type Queue

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

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

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

aaa  
bbb  
ccc  
ddd  
eee  
fff  
fin

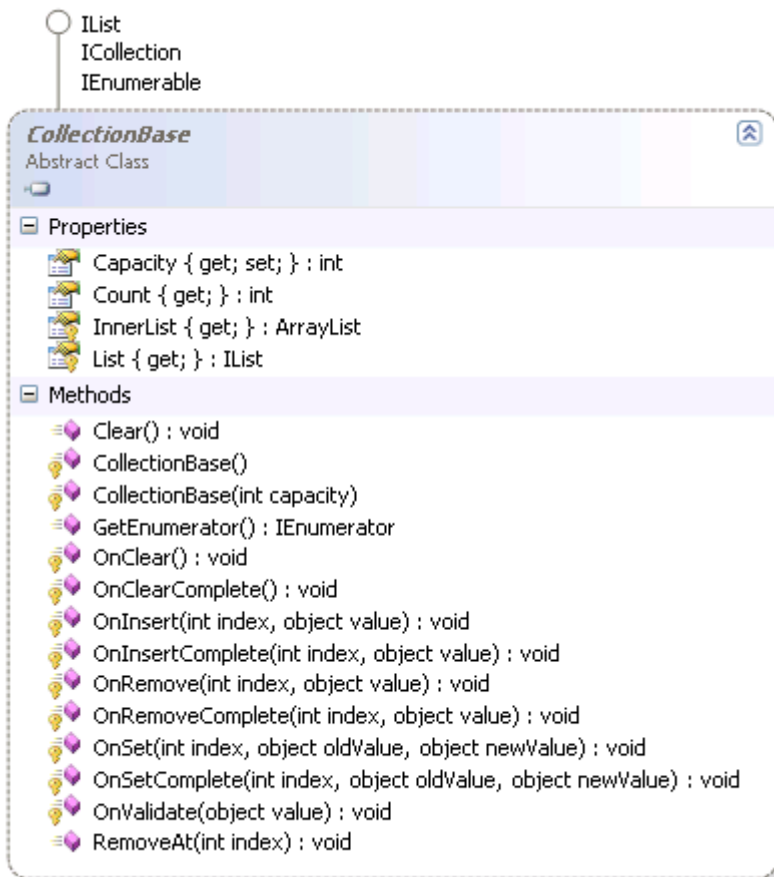


## Classes de bases pour collections personnalisées

### CollectionBase

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

La classe `CollectionBase` :

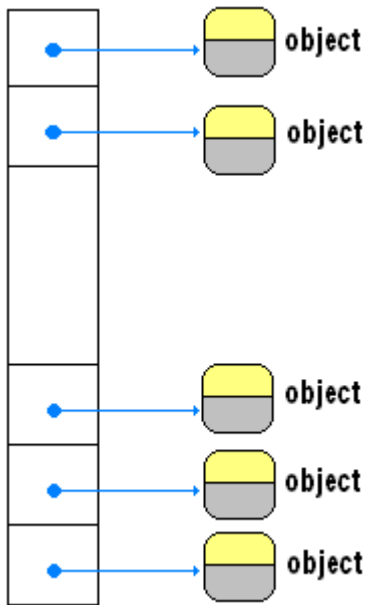


### Les interfaces implémentées par CollectionBase

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



## Schéma interne d'un objet de classe `CollectionBase` :



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

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

```
class MaCollection : CollectionBase { ... }
```

## DictionaryBase

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

### Les interfaces implémentées par `DictionaryBase`

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

IDictionary  
 ICollection  
 IEnumerable

**DictionaryBase**  
Abstract Class

**Properties**

- Count { get; } : int
- Dictionary { get; } : IDictionary
- InnerHashtable { get; } : Hashtable

**Methods**

- Clear() : void
- CopyTo(Array array, int index) : void
- DictionaryBase()
- GetEnumerator() : IDictionaryEnumerator
- OnClear() : void
- OnClearComplete() : void
- OnGet(object key, object currentValue) : object
- OnInsert(object key, object value) : void
- OnInsertComplete(object key, object value) : void
- OnRemove(object key, object value) : void
- OnRemoveComplete(object key, object value) : void
- OnSet(object key, object oldValue, object newValue) : void
- OnSetComplete(object key, object oldValue, object newValue) : void
- OnValidate(object key, object value) : void

**classe DictionaryBase**

**Schéma interne d'un objet de classe DictionaryBase :**

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

**IDictionary**  
Interface

- ICollection
- IEnumerable

**Properties**

- IsFixedSize { get; } : bool
- IsReadOnly { get; } : bool
- Item[object key] { get; set; } : object
- Keys { get; } : ICollection
- Values { get; } : ICollection

**Methods**

- Add(object key, object value) : void
- Clear() : void
- Contains(object key) : bool
- GetEnumerator() : IDictionaryEnumerator
- Remove(object key) : void

Keys  
ICollection

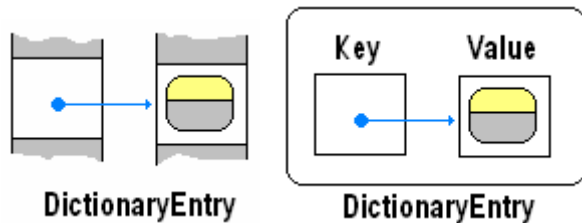
Values  
ICollection

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

```
class MonDictionnaire: DictionaryBase { ... }
```

### Accès aux données en lecture seulement directement :

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



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

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

### Accès aux données en lecture seulement à travers le Dictionary :

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

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

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

### Exemple de classe dérivée de DictionaryBase :

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

```

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

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

```

Résultats obtenus :

```

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

```