

LE LANGAGE C#

Introduction au framework .NET

RECOMMANDATIONS

La maîtrise d'un langage passe par une **pratique importante**.

Vous devez:

- Lire et relire le cours en vous appuyant sur la biblio donnée ci après
- reprendre les exemples du poly dans Visual Studio (copier/coller), les exécuter et les étudier.
- Vous devez FAIRE les exercices d'auto apprentissage

BIBLIOGRAPHIE ET RESSOURCES

Livres

Oreilly.CSharp.4.0.in.a.Nutshell.4th.Edition (le meilleur à mon avis)

- en anglais
- Une référence: très complet
- Nécessite à mon avis quelques notions

CSharp.2010.and.the.dotNET.4.Platform.5th.Edition:

- en anglais
- Bien

C# et .Net - Versions 1 à 4:

- En français
- Bon livre pour les débutants
- Moins complet que **CSharp.4.0.in.a.Nutshell**

APPRENTISSAGE DU LANGAGE C# 2008 et du Framework .NET 3.

- Poly de cours de C# par Serge Tahé
- En français

Pratique de .NET et C# (patrick SMACCHIA)

- En français
- Mal structuré à mon avis: partie 1 du livre très technique (informations que vous ne trouverez pas dans les autres livres) et partie 2: Très bien

Internet

[http://msdn.microsoft.com/fr-fr/library/ms229335\(v=VS.80\).aspx](http://msdn.microsoft.com/fr-fr/library/ms229335(v=VS.80).aspx)

<http://www.developpez.com/>

www.codeproject.com/

<http://www.codeguru.com/>

Sur les design pattern des articles très didactiques:

http://www.siteduzero.com/tutoriel-3-10601-programmation-en-java.html#part_65564
et tous les autres.....

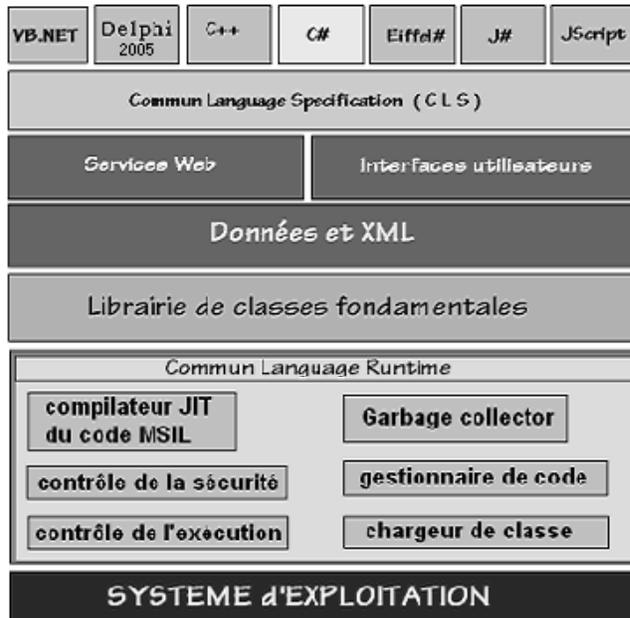
INTRODUCTION

L'objectif de ce polycopié n'est pas une présentation exhaustive du langage C#. Pour cela une littérature nombreuse existe sur Internet (en version papier aussi d'ailleurs). Le C# peut être considéré comme une synthèse du C++ et de Java. Tout en gardant les points forts du C++, il corrige certains points faibles et permet une abstraction de données plus poussée et une approche complètement orientée objet. Toutefois certains inconvénients existent à savoir :

Un langage qui reste en partie « propriétaire »

Langage Non temps réel (ramasse miette non déterministe, langage intermédiaire interprété)

La plate forme .NET Framework



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.

La seconde couche est un ensemble de composants graphiques disponibles dans Visual Studio .NET

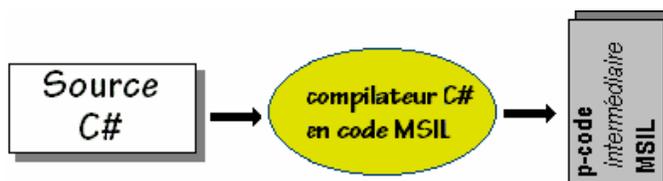
La troisième couche est constituée d'une vaste librairie de plusieurs centaines de classes

La quatrième couche forme l'environnement d'exécution commun (CLR ou CommonLanguage 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

L'environnement d'exécution du CLR

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.

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**. Cela signifie que **vous ne pourrez pas l'exécuter directement** sur un ordinateur qui n'aurait pas la machine virtuelle.NET.



Lorsque le processeur P n'est pas une machine qui existe physiquement mais un logiciel simulant (ou interprétant) une machine on appelle cette machine **pseudomachine** 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 compilation JIT progressive

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.

La compilation AOT

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.

ELEMENTS DE BASES DU C#

Les données natives

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

ATTENTION : le `char` est un nombre entier non signé **sur 2 octets**

Nature des objets

Le système de types C# comporte les catégories suivantes :

- Types valeur
- Types référence
- (Types de pointeur)

Les variables de types valeur stockent des données alors que les variables de types référence stockent les *références* aux données. Les types référence sont également considérés comme des objets. Les types pointeur ne peuvent être utilisés qu'en mode **non sécurisé** et sont donc à éviter.

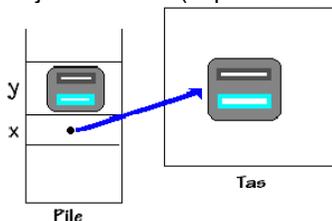
Contrairement au C++ le programmeur ne maîtrise pas l'emplacement des objets en mémoire et que partiellement leur durée de vie (c'est le ramasse miette qui est le maître d'œuvre du cycle de vie des objets).

L'emplacement des objets en mémoire dépendra de plusieurs paramètres:

- de leur nature :type valeur ou type référence
- de la durée vie de l'objet
 - o pour des durées courtes (registre processeur ou pile)
 - o pour des durées longues: le TAS

Les variables

Les variables basées sur des types valeur contiennent directement des valeurs. L'attribution d'une variable de type valeur à une autre, copie la valeur contenue,d'où le terme "type valeur" (copie par valeur) Cela est différent de l'assignation des variables de type référence, qui copient une référence vers l'objet mais pas l'objet lui-même (copie des références).



y est de « type » variable. Le contenu de la variable reflète le contenu de la variable y

x est de type référence. Le contenu de la variable contient "un lien" vers la donnée situé ailleurs en mémoire.

Le tas est en fait un espace de mémoire vive disponible pour le processus courant, c'est-à-dire l'ensemble de votre programme (et de ses *thread*, s'il y en a plusieurs). La pile est une zone mémoire privée allouée pour chaque *thread*.

```
//types flottant
float y = 4.5f;
short z = 5;
double w = 1.7E+3;
// declaration structure
struct StructAmoi {
    int b;
    void meth(int a){
        b = 1000+a;
    }
}
// instantiation structure
StructAmoi y = new StructAmoi ();
```

Les types "valeur" sont répartis en deux catégories principales :

- Struct
- Énumérations

Les struct sont répartis dans les catégories suivantes :

- ✓ Types numériques
- ✓ Types entiers (des entiers)
- ✓ Types virgule flottante
- ✓ decimal
- ✓ bool
- ✓ Structures définies par l'utilisateur.

Les références

Précision sur la notion de référence:

- Une référence établit "un lien" vers l'objet visé
 - o Attention: une référence *n'est pas l'adresse de l'objet référencé*. Il peut s'agir par exemple d'un index ou identificateur spécifique maintenu par le ramasse miette et /ou le CLR.
- Une référence ne donne pas accès à l'adresse physique de l'objet
- Les opérations arithmétiques sont interdites
- Une référence doit être « initialisée » à la création
- Une référence est nullable (`string chaine=NULL ;`)
- Une référence permet d'accéder au contenu de l'objet visé et de le modifier
- Une référence est de type variable: A l'instant la référence est connecté à l'objet A puis à t+1 à l'objet B.

Les mots clés suivants induisent la création d'une variable de type référence

- class
- interface
- delegate
- object
- string

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine((object)a == (object)b);
```

sans oublier les objets tableaux et chaînes.

Exemple:

```
class TypeRefExemple
{
    public float note;
    public string nomEtudiant;
}
.....
TypeRefExemple Ref1;// allocation d'une référence (non initialisé!)
TypeRefExemple Ref1=new TypeRefExemple(); // création d'un objet du type TypeRefExemple , Ref1 "pointe" vers l'objet
TypeRefExemple Ref2=Ref1; // création d'une nouvelle référence "pointant" vers l'objet précédent
```

Req: *Ref1, et Ref2 se retrouve dans la pile du thread courant*
L'objet référencé est placé dans la TAS
Nous avons donc 2 références mais un seul objet de type TypeRefExemple

Structure d'un programme type console

```
using System;
```

```
namespace date_test_minute
{
    class Program
    {
        static void Main(string[] args)
        {
            // vos lignes de codes ici
        }
    }
}
```

Commentaires :

using: permet de rajouter des composants logiciels
namespace: permet de construire ses propres distributions de classes
class: Une classe permet de regrouper au sein d'une même structure le format des données qui définissent l'objet et les fonctions destinées à manipuler ces objets. On dit qu'une classe permet d'encapsuler les différents éléments et les fonctions dans un ensemble appelé objet

Un exemple à étudier attentivement sur la création d'objets (*lisez les commentaires!*):

```
using System;

namespace Exemple_valeur_reference
{
    struct TypeValExemple
    {
        public int code_Apoge;
        public string nomEtudiant;
    }

    class TypeRefExemple
    {
        public float note;
        public string nomEtudiant;
    }

    class Program
    {
        static void Main()
        {
            string nom_1="Dupont"; //type référence
            float note=13.5f; //type valeur
            int code=12321; //type valeur
            string nom_2; //type référence
            // creation de deux variables du type TypeValExemple
            TypeValExemple Etudiant1_TypeVal; // forme 1
            TypeValExemple Etudiant2_TypeVal=new TypeValExemple();// forme 2 d'allocation

            TypeRefExemple Etudiant3_TypeRef1;// référence non initialisée
            TypeRefExemple Etudiant3_TypeRef2=new TypeRefExemple(); // fait référence au nouvel objet crée
            par l'opérateur new

            //la ligne suivante ne compile pas:Etudiant1_TypeVal est un objet non initialisé!!!
            //Console.WriteLine("Etud1 :"+Etudiant1_TypeVal.code_Apoge+"::"+Etudiant1_TypeVal.nomEtudiant);

            Etudiant2_TypeVal.code_Apoge = code;
            Etudiant2_TypeVal.nomEtudiant = nom_1;
            Etudiant1_TypeVal=Etudiant2_TypeVal; //2 objets séparés: copie en valeur champ par champ
            Console.WriteLine("Etud1 :"+Etudiant1_TypeVal.code_Apoge+"//"+Etudiant1_TypeVal.nomEtudiant);
            Console.WriteLine("Etud2      :"+      +      Etudiant2_TypeVal.code_Apoge      +      "///"      +
Etudiant2_TypeVal.nomEtudiant);
            nom_2 = nom_1;
            // ne pas confondre la variable note avec le champs note de l'objet Etudiant4_TypeRef de type
            TypeRefExemple
            Etudiant3_TypeRef2.note = note;
            Etudiant3_TypeRef2.nomEtudiant = nom_2;

            // la ligne suivante ne compile pas:TypeRefExemple n'est pas un objet( ie une variable) mais un
            type de données
            //TypeRefExemple.nomEtudiant = "test";

            // la ligne suivante ne compile pas:Etudiant3_TypeRef ne fait référence pour l'instant à aucun
            objet
            //Etudiant3_TypeRef1.nomEtudiant = "test";

            Etudiant3_TypeRef1 = Etudiant3_TypeRef2;// 1 seul objet en mémoire mais 2 références
            Console.WriteLine("Etud : {0} // {1}", Etudiant3_TypeRef1.note, Etudiant3_TypeRef1.nomEtudiant);
            // la note de l'étudiant 3 est modifiée
            Etudiant3_TypeRef1.note = 20;
            // ce changement apparait aussi sur l'etudiant4
            Console.WriteLine("Etud : {0} // {1}", Etudiant3_TypeRef2.note, Etudiant3_TypeRef2.nomEtudiant);
            // NORMAL: 1 seul objet mais 2 référence à ce même objet!

            Console.ReadKey();// attente touche
        }
    }
}
```

Flux d'entrée/sortie standard

La classe `System.Console` donne accès aux opérations d'écriture écran (**Write**, **WriteLine**) et de lecture clavier (**Read**, **ReadLine**). La classe `Console` a 3 propriétés **In**, **Out** et **Error** qui sont des flux d'écriture de type `TextWriter`.

- `Console.WriteLine()` est équivalent à `Console.Out.WriteLine()` et écrit sur le flux **Out** associé habituellement à l'écran.
- `Console.Error.WriteLine()` écrit sur le flux **Error**, habituellement associé lui aussi à l'écran.

Attention : La méthode `ReadLine` renvoie

Les formateurs

Ils sont basés sur une chaîne de caractère

`Console.Write (String, Object, Object)` où `String` est une chaîne type formateur

Formateur de nombres

Specifier	Type	Format	Output (Passed Double 1.42)	Output (Passed Int - 12400)
c	Currency	{0:c}	\$1.42	-\$12,400
d	Decimal (Whole number)	{0:d}	System.FormatException	-12400
e	Scientific	{0:e}	1.420000e+000	-1.240000e+004
f	Fixed point	{0:f}	1.42	-12400.00
g	General	{0:g}	1.42	-12400
n	Number with commas for thousands	{0:n}	1.42	-12,400
r	Round trippable	{0:r}	1.42	System.FormatException
x	Hexadecimal	{0:x4}	System.FormatException	d90

```
string myName = "Fred";
Console.WriteLine ("Name = {0}, hours = {1:hh}", myName,
DateTime.Now);
// {...} indique le numéro de l'argument à afficher
// Name = Fred, hours = 05
Console.WriteLine("(X) Hexa:{0:X}\n (D) Decimal: {0:D}\n", 255);
// (X) Hexa:80
// (D) Decimal: 128
Double myDouble = 1234567890;
Console.WriteLine("{0:###} ### - ####", myDouble);
// The value of myString is "(123) 456 - 7890".
```

Formateur de dates

Specifier	Type	Example (Passed System.DateTime.Now)
d	Short date	10/12/2002
D	Long date	December 10, 2002
t	Short time	10:11 PM
T	Long time	10:11:29 PM
f	Full date & time	December 10, 2002 10:11 PM
F	Full date & time (long)	December 10, 2002 10:11:29 PM
g	Default date & time	10/12/2002 10:11 PM
G	Default date & time (long)	10/12/2002 10:11:29 PM
M	Month day pattern	December 10
r	RFC1123 date string	Tue, 10 Dec 2002 22:11:29 GMT
s	Sortable date string	2002-12-10T22:11:29
u	Universal sortable, local time	2002-12-10 22:13:50Z
U	Universal sortable, GMT	December 11, 2002 3:13:50 AM
Y	Year month pattern	December, 2002

```
int i;
char c;
while (true)
{
    i = Console.Read ();
    if (i == -1) break;
    c = (char) i;
    Console.WriteLine ("Echo: {0}", c);
}
Console.WriteLine ("Done");
return 0;
```

Formateur spécifiques

Specifier	Type	Example	Output (Passed Double 1500.42)	Note
0	Zero placeholder	{0:00.0000}	1500.4200	Pads with zeroes.
#	Digit placeholder	{0:(#).##}	(1500).42	
.	Decimal point	{0:0.0}	1500.4	
,	Thousand separator	{0:0,0}	1,500	Must be between two zeroes.
,	Number scaling	{0:0,}	2	Comma adjacent to Period scales by 1000.
%	Percent	{0:0%}	150042%	Multiplies by 100, adds % sign.
e	Exponent placeholder	{0:00e+0}	15e+2	Many exponent formats available.
;	Group separator	see below		

Exemples complémentaires:

Saisie clavier avec conversion

`string -> float`

```
static void Main(string[] args)
{
    float a, b, c; // coeff equation
    Console.Write("Entrer a=");
    a = float.Parse(Console.ReadLine());
    Console.Write("Entrer b=");
    b = float.Parse(Console.ReadLine());
    Console.WriteLine("Entrer c=");
    c = float.Parse(Console.ReadLine());
}
```

Sortie Clavier

```
using System;
public class Example
{
    public static void Main()
    {
        string line;
        Console.WriteLine("Enter one or more lines
of text (press CTRL+Z to exit):");
        Console.WriteLine();
        do {
            Console.Write(" ");
            line = Console.ReadLine();
            if (line != null)
                Console.WriteLine(" " + line);
        } while (line != null);
    }
}
// The following displays possible output from
// this example:
// Enter one or more lines of text (press
// CTRL+Z to exit):
// This is line #1.
// This is line #1.
// This is line #2
// This is line #2
// ^Z
```

Les différentes structures de bases

Opérateurs

Bit à Bit

Soient i et j deux entiers.

- ✓ $i << n$ décale i de n bits sur la gauche. Les bits entrants sont des zéros.
- ✓ $i >> n$ décale i de n bits sur la droite. Si i est un entier signé (signed char, int, long) le bit de signe est préservé.
- ✓ $i \& j$ fait le ET logique de i et j bit à bit.
- ✓ $i | j$ fait le OU logique de i et j bit à bit.
- ✓ $\sim i$ complémente i à 1

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

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

Logique:

Idem langage C <, <=, ==, !=, >, >=

Arithmétique

Idem langage C +, -, \, %

Utilisation de la classe Math : exemple de méthode accesible

double Sqrt(double x) racine carrée
double Cos(double x) Cosinus
double Sin(double x) Sinus
double Tan(double x) Tangente
double Pow(double x, double y) x à la puissance y (x>0)
double Exp(double x) Exponentielle
double Log(double x) Logarithme népérien
double Abs(double x) valeur absolue

```
double x, y=4;
x=Math.Sqrt(y);
```

booléens

Idem langage C

Structure de choix

simple

syntaxe : if (condition) {actions_condition_vraie;}
else {actions_condition_fausse;}

cas multiples

```
switch(expression) {  
  case v1:actions1;  
  break;  
  case v2:      actions2;  
  break;  
  default:actions_sinon;
```

Structure de répétition

Identique au C

- for
- do/while
- while

Structure **foreach**

La syntaxe est la suivante :

```
foreach (Type variable in collection)  
{instructions;  
}
```

Notes

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

Ainsi le code suivant :

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

afficherait :
paul
hélène
jacques
sylvie

Chaîne de caractère

Codage d'un caractère

Un ordinateur ne comprenant que des valeurs numériques il est nécessaire d'associer à chaque symbole représentant un caractère (lettre de l'alphabet, symbole mathématique, musical etc) une valeur. Cette association symbole/valeur est appelé codage.

Ci après les codages les plus usuels:

- ASCII
 - o Le plus ancien. Permet de coder 128 symboles correspondant aux claviers US.
 - 1 symboles= 1 octet
- Code Page OEM
 - o Extension du code ASCII sur 256 octets pour tenir compte des langues européennes
- GB18030
 - o Standart pour les applications développées en Chine
- Unicode
 - o Organisation en 17 pages contenant 65536 codes possibles(1Mega codes possibles!)
 - o La page 0 est appelée " Basic Multilingual Plane" permettant de coder tous les langages modernes
 - o 4 variantes de codages dont:
 - UTF8 : le plus utilisé pour les fichiers textes et internet.
 - Codage variable de 1 à 4 octets
 - Les 127 premiers codes sont ceux de l'ASCII (d'où sa popularité)

- UTF16: codage sur 2 ou 4 octets
 - Codage par défaut utilisé pour les caractères en C#
 - 1 char = 2 octets => permet de représenter le BPM
 - Moins compact que l'UTF8 mais permet une manipulation plus aisée des flux de données(taille fixe du codage!)

Exemple d'utilisations d'outils pour les conversions:

```
byte[] utf8Bytes = System.Text.Encoding.UTF8.GetBytes ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789"); // Unicode correspond à UTF 16
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes ("0123456789");
Console.WriteLine (utf8Bytes.Length); // 10
Console.WriteLine (utf16Bytes.Length); // 20
Console.WriteLine (utf32Bytes.Length); // 40
string original1 = System.Text.Encoding.UTF8.GetString (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString (utf32Bytes);
Console.WriteLine (original1); // 0123456789
Console.WriteLine (original2); // 0123456789
Console.WriteLine (original3); // 0123456789
```

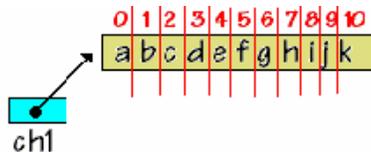
Les chaînes de caractères

Classe string

La classe string facilite les opérations de manipulations de chaîne (abandonnez le char !).

Représentation d'un objet string :

```
String ch1 = "abcdefghijk";
char car = ch1[4];
// ici la variable car contient la lettre 'e'
// pas de ' \0 ' à la fin !!
```



Extrait des méthodes accessibles

Methods

	Name	Description
⇒ X	Clone	Returns a reference to this instance of String .
⇒ S X	Compare	Overloaded. Compares two specified String objects and returns an integer that indicates their relationship to one another in the sort order.
⇒ S X	CompareOrdinal	Overloaded. Compares two String objects by evaluating the numeric values of the corresponding Char objects in each string.
⇒ S X	CompareTo	Overloaded. Compares this instance with a specified object or String and returns an integer that indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified object or String .
⇒ S X	Concat	Overloaded. Concatenates one or more instances of String , or the String representations of the values of one or more instances of Object .
⇒ S X	Contains	Returns a value indicating whether the specified String object occurs within this string.
⇒ S X	Copy	Creates a new instance of String with the same value as a specified String .
⇒ S X	CopyTo	Copies a specified number of characters from a specified position in this instance to a specified position in an array of Unicode characters.
⇒ S X	EndsWith	Overloaded. Determines whether the end of an instance of String matches a specified string.
⇒ S X	Equals	Overloaded. Determines whether two String objects have the same value.

Exemples

```
string ch;
ch = "abcdef" ;
string s2,s1="abc" ;
s2 = s1+"def";
//-- tests d'égalité avec l'opérateur ==
if( s2 == "abcdef" )
System.Console.WriteLine
("s2==abcdef");
else System.Console.WriteLine
("s2<>abcdef");
if( s2 == ch ) System.Console.WriteLine
("s2==ch");
else System.Console.WriteLine
("s2<>ch");
```

```
String s1 , s2 ="abc" ;
char c = 'e' ;
s1 = "d" + Convert.ToString ( c );
Console.WriteLine("chaîne{0}, taille{1}",s1,S1.lenght)
```

```

using System;

public class SplitTest {
    public static void Main() {
        string words = "This is a list of words, with: a bit of punctuation."
        string [] split = words.Split(new Char [] { ' ', ',', '.', ':' });
        foreach (string s in split)
            if (s.Trim() != "")
                Console.WriteLine(s);}}}}
// The example displays the following output to the console:
// This
// is
// a
// list
// of
// words
// with
// a
// bit
// of
// punctuation

```

Conversions Char<-> String

1- Char-> String

Plusieurs possibilités:

Sol1: Transtypage

```

char car = 'r';
string s;
s = (string)car;

```

Sol2: Méthode string.ToString

```

char car = 'r';
string s;
s = Convert.ToString(car);

```

2- String-> Char

Sur un ensemble de caractere

```

string input = "Hello World!";
char[] values = input.ToCharArray();

```

sur 1 caractère

```

char charVal = 'a';
stringVal = System.Convert.ToString(charVal);

```

Les Tableaux

Un tableau est une structure de données qui contient plusieurs variables du même type. Les tableaux sont déclarés avec un type :

```
type[] arrayName;
```

Les exemples suivants créent des tableaux unidimensionnels, multidimensionnels et en escalier :

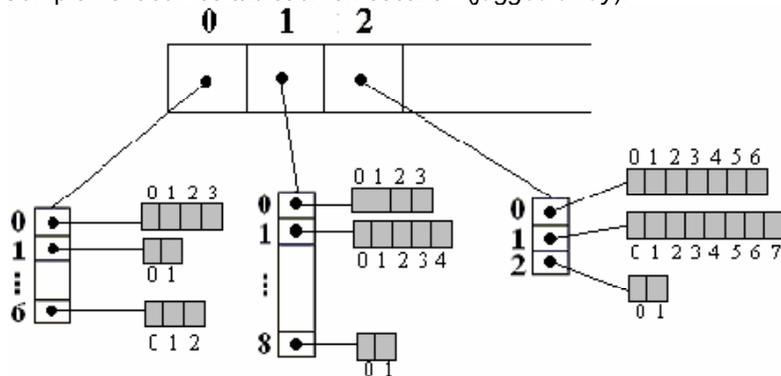
```

// Declare a single-dimensional array
int[] array1 = new int[5];
// Declare and set array element values
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
// Alternative syntax
int[] array3 = { 1, 2, 3, 4, 5, 6 };
// Declare a two dimensional array
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
// Declare a jagged array
int[][] jaggedArray = new int[6][];
// Set the values of the first array in the jagged array structure
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };

```

Complément sur les tableaux en escalier (*jagged array*):



Les liens s'expriment par des références sur tableau (le contenu d'une case est une référence qui « pointe » vers un tableau)

Formatage et conversion des données

Certaines instructions permettent de gérer des flux d'entrées/sorties formatés permettant ainsi un échange d'information aisé entre variable (l'échange ne se fait pas avec le monde extérieur).

Un exemple classique de fonction en C est : `sprintf()` et `scanf`

Les outils en C# sont :

Formatage numérique vers string

Syntaxe: `xxxx.ToString(chaîne de formatage)`

```
double MyPos = 19.95 ,MyNeg = -19.95, MyZero = 0.0;
string MyString = MyPos.ToString("$#,##0.00;($#,##0.00);Zero"); // $19,95
MyString = MyNeg.ToString("$#,##0.00;($#,##0.00);Zero"); // ($19,95)
MyString = MyZero.ToString("$#,##0.00;($#,##0.00);Zero"); // Zero
```

Formatage string vers numérique

chaîne -> int : `int.Parse(chaîne)` ou `System.Int32.Parse`
 chaîne -> long : `long.Parse(chaîne)` ou `System.Int64.Parse`
 chaîne -> double : `double.Parse(chaîne)` ou `System.Double.Parse(chaîne)`
 chaîne -> float : `float.Parse(chaîne)` ou `System.Float.Parse(chaîne)`

```
String num = "A";
int val = int.Parse(num, NumberStyles.HexNumber);
Console.WriteLine("{0} in hex = {1} in decimal.", num, val);

// Parse the string, allowing a leading sign, and ignoring leading
and trailing white spaces.
num = " -45 ";
val = int.Parse(num, NumberStyles.AllowLeadingSign |
    NumberStyles.AllowLeadingWhite |
    NumberStyles.AllowTrailingWhite);
Console.WriteLine("'0' parsed to an int is '{1}'.", num, val);
```

Formatage composite

numérique + string -> string

Syntaxe : `String.Format` (paramètre chaine de formatage voir classe Console)

```
string myName = "Fred";
String.Format("Name = {0}, hours = {1:hh}, minutes = {1:mm}",
    myName, DateTime.Now);
```

```
string FormatString=String.Format("{0:dddd MMMM}", DateTime.Now);
string FormatString2 = DateTime.Now.ToString("dddd MMMM");
```

string
g ->

string +numérique

Ce type de formatage correspond au sscanf du C. La fonction C# n'existe pas en tant que telle et doit être réalisée en fonction des besoins de l'utilisateur à l'aide des méthodes: `.Split`, `.Parse` `.Substring`

```
string[] Command = { "LED1 ON", "TEMP 25", "VIT,G,100" };
char [] separateur={' ',';'};

for (int i = 0; i < Command.Length; i++)
{
    string oneCmd = Command[i];
    string[] mot = oneCmd.Split(separateur);
    // affichage des mots
    foreach (string v in mot)
        Console.WriteLine(v);
    // traitement
    if (mot[0] == "TEMP")
    {
        int temp = int.Parse(mot[1]);
        Console.WriteLine("temp consigne {0}", temp);
    }
}
```

TRAVAIL PERSONNEL

Pour faire une pause à la fin de votre programme - terminer par exemple par `Console.ReadKey();`

Exo1 :

On souhaite écrire un programme C# de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$

Spécifications de l'algorithme :

Algorithme Equation
Entrée: A, B, C ∈ Réels
Sortie: X1, X2 ∈ Réels
Local: Δ ∈ Réels

début
lire(A, B, C);
Si A=0 alors début{A=0}
 Si B = 0 alors
 Si C = 0 alors
 écrire(R est solution)
 Sinon {C ≠ 0}
 écrire(pas de solution)
 Fsi
 Sinon {B ≠ 0}
 X1 ← C/B;
 écrire (X1)
 Fsi
fin
Sinon {A ≠ 0} début

```
Δ ← B2 - 4*A*C ;  
Si Δ < 0 alors  
  écrire(pas de solution)  
Sinon {Δ ≥ 0}  
  Si Δ = 0 alors  
    X1 ← -B/(2*A);  
    écrire (X1)  
  Sinon {Δ ≠ 0}  
    X1 ← (-B + √Δ)/(2*A);  
    X2 ← (-B - √Δ)/(2*A);  
    écrire(X1 , X2 )  
  Fsi  
Fsi  
fin  
Fsi  
FinEquation
```

Ex2

Dans une boucle le prix d'un article hors taxe est saisi. Le calcul du prix TTC sera calculé puis seront affichés le prix final sous forme "monnaie" (2 chiffres après virgule, voir formatage) ainsi que le taux de la TVA en pourcent (1 chiffre après la virgule).

On sortira de la boucle sur une demande de nouveau calcul .

Exo 3

Objectif : compter et afficher le nombre de voyelles compte tenu dans une chaîne de caractères.

Un chaîne contenant les voyelles sera utilisé : `string voyelle="...."`.

Afin de vous aider faites des recherches sur internet pour trouver les méthodes de la classe string pour:

- convertir en majuscules
- tester si une chaîne contient une lettre (ou une sous-chaîne)

Exo4

Une date sera saisie sous la forme JJ/MM/AAAA (pas de vérification sur la cohérence de la saisie). Les délimiteurs possibles sont: / et - (exemple:JJ-MM-AAAA est valide)

Extraire en utilisant les méthodes de la classe string les différents champs de la date saisie et les stocker dans 3 variables de type entier.

Pour l'exercice les boucles foreach sont imposées pour balayer vos tableaux).

TP

Exo 1

Coder l'algorithme de tri à bulle suivant:

Spécifications de l'algorithme :

```
Algorithme Tri a Bulles
local: i, j, n, temp ∈ Entiers naturels
Entrée - Sortie : Tab ∈ Tableau d'Entiers naturels de 1 à n éléments
début
pour i de n jusqu'à 1 faire // recommence une sous-suite (a1, a2, ..., ai)
pour j de 2 jusqu'à i faire // échange des couples non classés de la sous-suite
si Tab[j-1] > Tab[j] alors // aj-1 et aj non ordonnés
temp ← Tab[j-1];
Tab[j-1] ← Tab[j];
Tab[j] ← temp //on échange les positions de aj-1 et aj
Fsi
fpour
fpour
Fin Tri_a_Bulles
```

Exo2 :

Ecrire un programme calculant sur 360 degré n points de la fonction sinus . Le paramètre n sera saisi clavier. Les points seront stockés au préalable dans un tableau qui sera par la suite parcouru pour affichage (instruction foreach imposée).

i=5 angle=225,0 val=-0,71

i=6 angle=270,0 val=-1,00

i=7 angle=315,0 val=-0,71

Exo 3

Ecrire la méthode **static string** toBinaryString (int n) qui permet de visualiser le code binaire sous forme de chaîne de caractère d'un nombre entier n codé en int.

Pour ce faire vous pourrez exploitez les possibilités de la méthodes Format (conversion int->hexa) et IndexOf

les tableaux suivant

```
string [ ] hexToBinDigit = { "0000", "0001", "0010", "0011", "0100",
                             "0101", "0110", "0111", "1000", "1001", "1010",
                             "1011", "1100", "1101", "1110", "1111" };
string hexToDecimalDigit="0123456789abcdef";
```

Exo4

On souhaite écrire une application de manipulation interne des bits d'une variable signée sur 32 bits (int). Le but de l'exercice est de construire une famille de méthodes de travail sur un ou plusieurs bits d'une mémoire. La visualisation des résultats se fera à l'aide de la méthode toBinaryString précédemment créée.

L'application à construire contiendra 9 méthodes :

1. Une méthode **BitSET** permettant de mettre à **1** un bit de rang fixé. `static int BitSET (int nbr, int num) { }`
2. Une méthode **BitCLR** permettant de mettre à **0** un bit de rang fixé. `static int BitCLR (int nbr, int num) { }`
3. Une méthode **BitCHG** permettant de remplacer un bit de rang fixé par son complément. `static int BitCHG (int nbr, int num) { }`
4. Une méthode **SetValBit** permettant de modifier un bit de rang fixé. `static int SetValBit (int nbr, int rang, int val) { }`
5. Une méthode **DecalageD** permettant de décaler les bits d'un entier, sur la droite de n positions (introduction de n zéros à gauche). `static int DecalageD (int nbr, int n) { }`
6. Une méthode **DecalageG** permettant de décaler les bits d'un entier, sur la gauche de n positions (introduction de n zéros à droite). `static int DecalageG (int nbr, int n) { }`
7. Une méthode **BitRang** renvoyant le bit de rang fixé d'un entier. `static int BitRang (int nbr, int rang) { }`
8. Une méthode **ROL** permettant de décaler avec rotation, les bits d'un entier, sur la droite de n positions (réintroduction à gauche). `static int ROL (int nbr, int n) { }`
9. Une méthode **ROR** permettant de décaler avec rotation, les bits d'un entier, sur la gauche de n positions (réintroduction à droite). `static int ROR (int nbr, int n) { }` Les méthodes **ROL** et **ROR** réutiliseront les autres méthodes (réinjection du bit de poids fort sur le faible)

Proposition de squelette de classe C# à implanter :

```
class Application8Bits {
static void Main(string [ ] args){
..... }
static int BitSET (int nbr, int num) { ..... }
static int BitCLR (int nbr, int num) { ..... }
.....
}
```

Observez le codage d'entiers négatifs. Comment s'appelle ce codage?

Quel est l'effet d'un décalage à gauche d'un entier négatif? Expliquez.

NOTION D'OBJET

La complexité du monde réel nous conduit à définir, classifier, organiser et hiérarchiser les différents objets qui le composent.

Ces objets possèdent des propriétés et des comportements propres. Une même famille d'objets, c'est-à-dire avec des caractéristiques communes, peut être regroupée dans un ensemble appelé classe.

La programmation orientée objet reprend à son compte ces notions d'objet et de classe. Le développeur pourra à sa guise créer une classe, définissant les caractéristiques et les propriétés d'une même famille d'objet. Un objet, c'est-à-dire à un élément particulier d'une classe, est appelé instance d'une classe.

Exemples :

Toutou
propriétés :
âge, poids, couleur
comportements :
courir, aboyer, manger

Etudiant
Propriétés :
Nom, âge
Méthodes
bavarder, écouter, dormir

L'objectif est de faciliter et de fiabiliser l'écriture de programmes complexes en décomposant l'analyse à l'aide de différentes entités distinctes (classe, objet).

Du point de vue interne, l'objet est décrit en détail tant au niveau des propriétés que de ses méthodes d'action.

Du point de vue externe, l'objet est une boîte noire accessible au travers d'une certaine interface (les méthodes).

Avantages :

- réutilisation des objets
- structuration modulaire
- autonomie des objets
- maintenance et debuggage facilités

UTILISATION DES CLASSES ET MANIPULATIONS D'OBJETS

Définition d'une classe

Une classe permet de regrouper au sein d'une même structure le format des données qui définissent l'objet et les fonctions destinées à manipuler ces objets. On dit qu'une classe permet d'encapsuler les différents éléments et les fonctions dans un ensemble appelé objet.

//Version 1 :1 seul fichier

```
using System;
```

```
class Chat  
{  
    uint Age; // par défaut Private!!  
    private uint Poids;  
  
    public void Miauler()  
    { Console.WriteLine("Miaouuu"); }  
    public void DefAge(uint var)  
    { Age = var; }  
}
```

Déclaration de la classe

```
class Program
```

```
{  
    static void Main(string[]  
args)  
    {  
        Chat UnMatou;  
        UnMatou = new Chat();  
        UnMatou.Miauler();  
        UnMatou.DefAge(10);  
        Console.ReadKey();  
    }  
}
```

Utilisation
de la
classe

Commentaires

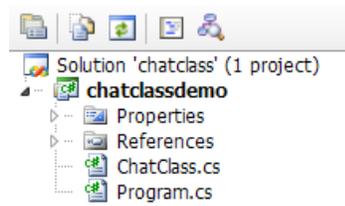
- ✓ Âge et poids correspondent aux champs de la classe (ie données, état interne)
- ✓ Miauler et DefAge correspondent à une méthode de la classe.
- ✓ Le mot-clé **public** permet de signifier que les méthodes ou propriétés sont accessibles de l'extérieur.
- ✓ Le mot-clé **private** permet de signifier que les données ne sont accessibles que par les méthodes appartenant à la classe (non accessible de l'extérieur directement).
- ✓ `Chat UnMatou;` crée une référence sur Chat

- ✓ UnMatou = new Chat(); crée une instance de Chat (ou objet de type de Chat)
- ✓ UnMatou.Age=10 ; ne compile pas car Age est private donc inaccessible de l'extérieur !

// version 2 : 2 fichiers

```
using System;
using ProjetChat;
```

```
class Program
{
    static void Main(
args)
    {
        Chat UnMatou;
        UnMatou = new Chat();
        UnMatou.Miauler();
        UnMatou.DefAge(10);
        Console.ReadKey();
    }
}
```



```
System;

namespace ProjetChat
{
    public class Chat
    {
        uint Age; // par défaut
        private uint Poids;

        public void Miauler()
        {
            Console.WriteLine("Miaouuu"); }
        public void DefAge(uint var)
        { Age = var; }
    }
}
```

Commentaires :

- ✓ La classe est livrée dans un fichier séparé pour une meilleure lisibilité
- ✓ L'espace de nommage étant différent, la clause using ProjetChat est utilisée pour la visibilité de la classe Chat

De manière générale et pour des raisons de sécurité les champs sont maintenus private et les méthodes publiques.

De cette manière, en cas de problème sur un objet, seules les méthodes appartenant à la classe incriminée peuvent être mises en cause puisque ce sont les seules qui puissent accéder aux données.

Création et vie des objets

L'opérateur new permet de créer les objets qui seront stockés en RAM. La référence renvoyée sert à initialiser la référence utilisateur pour la manipulation des objets :

```
Chat Matou1,Matou2; // 2 référence nulles ( non initialisées)
Matou1= new Chat();
Matou2= new Chat();
...
Matou1=new Chat(); // Matou1 fait référence a un 3ème chat!
```

Chaque objet en mémoire possède ses propres propriétés Age et Poids;

Au moment de la création de l'objet, une méthode spéciale, le constructeur, est appelée. Ce mécanisme répond à la question : *comment créer et initialiser l'objet.*

Un constructeur par défaut existe : exemple d'appel new Chat();

La fin de vie de l'objet est décidée par le ramasse-miette qui désalloue l'espace mémoire. Il en résulte que ,non contrôlé par l'utilisateur, ce mécanisme est non déterministe (ne convient pas pour des systèmes temps réels au sens strict).

Mise en oeuvre des méthodes des classes

Principe

Les méthodes des classes ne sont ni plus ni moins que des fonctions qui permettent :

- ✓ de définir certains comportements des objets
- ✓ d'accéder (interface publique) et de manipuler les champs attributs des objets

Les méthodes sont appelées au travers d'un objet : `UnMatou.Miauler()` ;

Lors de l'appel d'une méthode une référence sur l'objet courant (dans l'exemple `UnMatou`) est transmise implicitement par le biais d'un mot clé **this**.

Exemple : appel de la méthode: `UnMatou.SetPoids(5)` ;

```
class Chat
{
  uint Poids
  ....
  public void setPoids (uint Poids)
  { Poids = Poids; }
}
```

Commentaire :
Il y a ambiguïté entre `Poids` variable locale transmise lors de l'appel et `Poids` propriété de la Class

```
class Chat
{
  uint Poids
  ....
  public void setPoids (uint Poids)
  { this.Poids = Poids; }
}
```

Commentaire :
`This` : objet courant (`UnMatou`)
`This.Poids` : accès à la variable `Poids` de l'objet courant (`UnMatou`)

Solution: emploi de **this**

Passage d'argument

Passage par valeur

Si un paramètre est déclaré pour une méthode sans `ref` ni `out` le passage se fait par valeur : une variable ou une référence locale est créée puis la valeur recopiée.

```
class Program
{
  static void Method(string s2, MonInt test2, int varlocal)
  {
    s2 = "changed";
    test2.vartest = 10;
    varlocal = 10;
  }

  public class MonInt
  {
    public int vartest=5;
  }

  static void Main(string[] args)
  {
    string str = "original";
    MonInt testint = new MonInt();
    int var = 5;
    Method(str, testint, var);
    Console.WriteLine(str); // "original";
    Console.WriteLine(testint.vartest); // 10
    Console.WriteLine(var); // 5
    Console.ReadKey();
  }
}
```

- ✓ `Testint` est une classe donc `testint` est une référence.
- ✓ `Test2` est une copie locale de la référence (mais pas de l'objet)
- ✓ `test2.vartest = 10;` modifie la case mémoire
- ✓ `var` est créée dans la pile (objet valeur)
- ✓ `varlocal` est une copie locale=> pas de modification de `var`
- ✓ Le cas `string` est à part puisque bien que `str` soit une référence il y a création d'un nouvel espace mémoire et recopie du contenu pointé par `str`. La classe `string` est dite immuable (en anglais `immutable`)=> une fois initialisé par son constructeur toute modification entraîne un nouvel objet de type `string`

REMARQUE :Un tableau est toujours désigné par sa référence et est alloué par `new` (même si ce `new` n'apparaît pas quand un tableau est créé et initialisé dans la foulée). C'est à ce moment que la véritable zone de données du tableau est allouée. La référence « pointe » alors sur cette zone.

EN RESUME: UN PASSAGE PAR VALEUR CREE UNE COPIE TEMPORAIRE PAR CLONAGE:

- dans le cas d'un type 'value' : copie d'un objet temporaire ayant même valeur;
- dans le cas d'un type référence: clonage des références => il n'y pas création d'un nouvel objet!
Même si le mécanisme de passage s'est fait par valeur , du point de vue de l'objet, le passage s'est fait "par référence"

Passage par référence

Mot clé ref

Le mot clé **ref** fait en sorte que les arguments soient passés par référence. La conséquence est que toute modification apportée au paramètre dans la méthode est reflétée dans cette variable lorsque la méthode appelante récupère le contrôle. Pour utiliser un paramètre **ref**, l'appelant et l'appelé doivent tous deux utiliser le mot clé **ref** explicitement. Par exemple :

```
class RefExample1
{
    static void Method(ref int i)
    {
        i = 44;
    }
    static void Main()
    {
        int val = 0;
        Method(ref val);
        // val is now 44
    }
}
```

```
class RefRefExample2
{
    static void Method(ref string s)
    {
        s = "changed";
    }
    static void Main()
    {
        string str = "original";
        Method(ref str);
        // str is now "changed"
    }
}
```

Mot clé out

Le mot clé **out** fait en sorte que les arguments soient passés par référence. Il est semblable au mot clé **ref**, mais **ref** nécessite que la variable soit initialisée avant d'être passée. Pour utiliser un paramètre **out**, la définition de méthode et la méthode d'appel doivent toutes deux utiliser le mot clé **out** explicitement. Par exemple :

```
Class OutExample1
{
    static void Method(out int i)
    {
        i = 44;
    }
    static void Main()
    {
        int value;
        Method(out value);
        // value is now 44
    }
}
```

```
class OutReturnExample2
{
    static void Method(out int i, out string s1, out string s2)
    {
        i = 44;
        s1 = "I've been returned";
        s2 = null;
    }
    static void Main()
    {
        int value;
        string str1, str2;
        Method(out value, out str1, out str2);
        // value is now 44
        // str1 is now "I've been returned"
        // str2 is (still) null]}
}
```

Accès aux propriétés

Par les raisons déjà évoquées précédemment les propriétés des classes seront déclarée *private*. Un accès direct dans votre programme est interdit : `UnMatou.Poids=10`.

Des interfaces d'accès, nommées accesseurs (getter/setter in english) sont donc nécessaires pour des lectures/écritures simples.

Deux façons de procéder sont possibles :

1- accesseurs standarts

```
using System;

public class Chat
{
    uint Age; // par défaut Private!!

    public void Miauler()
    { Console.WriteLine("Miaouuu"); }
    // Accesseurs
    public void SetAge(uint var)
    { this.Age = var; }
    public uint GetAge()
    { return (this.Age); }
}

class Program
{
    static void Main(string[] args)
    {
        Chat UnMatou= new Chat();
        UnMatou.SetAge(10);
        Console.WriteLine("le matou a " +
UnMatou.GetAge()+" ans");
        Console.ReadKey();
    }
}
```

Les propriétés sont des membres offrant un mécanisme souple pour la lecture, l'écriture ou le calcul des valeurs de champs privés. Elles peuvent être utilisées comme si elles étaient des membres de données publiques, mais en fait, ce sont des méthodes spéciales appelées *accesseurs*. Elles facilitent l'accès aux données tout en préservant la sécurité et la souplesse des méthodes

- Avec des propriétés, une classe peut exposer de manière publique l'obtention et la définition de valeurs, tout en masquant le code d'implémentation ou de vérification.
- Un accesseur de propriété `get` permet de retourner la valeur de propriété, et un accesseur `set` est utilisé pour assigner une nouvelle valeur. Ces accesseurs peuvent avoir des niveaux d'accès différents. Pour plus d'informations, consultez [Accessibilité de l'accesseur asymétrique \(Guide de programmation C#\)](#).
- Le mot clé `value` sert à définir la valeur assignée par l'indexeur `set`.
- Les propriétés qui n'implémentent pas de méthode `set` sont en lecture seule.

2- Propriétés

```
using System;

public class Chat
{
    uint Age; // par défaut Private!!

    public void Miauler()
    { Console.WriteLine("Miaouuu"); }
    // Propriétés :get/set
    public uint AgeChat
    {
        get { return this.Age; }
        set
        {
            if (value < 0 || value > 30)
            {
                throw new Exception("Age du Chat
(" + value + ") invalide");
            }
            else
            {
                this.Age = value;
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Chat UnMatou= new Chat();
        UnMatou.AgeChat=10;
        Console.WriteLine("le matou a " +
UnMatou.AgeChat+" ans");
        //try-catch
        try
        {UnMatou.AgeChat = 35;
        }
        catch (Exception ex)
        {
            Console.Error.WriteLine(ex.Message);
        }
        Console.ReadKey();
    }
}
```

Remarque: La plupart des cas les "get et set" se bornent à des lectures/écritures simples. Le C# 3 introduit les propriétés automatiques (à déconseiller pour débuter).

EX: Class MaClasseDemo

```
{ public int age {get;set;} // Il a création automatique d'un champ privé correspondant à l'age
public string nom {get;set;} // Il a création automatique d'un champ privé correspondant au nom
}
```

Méthodes et champs statiques

Les Méthodes et champs statiques ne s'appellent pas au travers d'un objet comme les exemples précédents mais à partir de leur classe d'objet:

Méthode d'instance

```
MaClassObjet Objet1;  
Objet1.MaMethode;
```

Méthode statique

```
MaClassObjet.UneAutreMethode
```

Ci après des exemples de méthodes statiques déjà manipulées:

- Console.WriteLine() où WriteLine est la méthode statique de la classe Console.
- String.Parse() où Parse est la méthode statique de la classe string

L'attribut statique devant un champ d'une classe indique que la donnée (la variable) conserve sa valeur sur les différentes instances de la classe. Nous pouvons dire en quelque sorte que cette donnée sera partagée entre les différents objets issus de la même classe. Cette facilité permet notamment de compter le nombre d'instance d'une classe (ie nombre d'objets) et par exemple de contrôler une population donnée ou le nombre d'accès à une ressource.

Pour l'exemple suivant reportez vous au chapitre suivant sur les constructeurs:

```
using System;  
using System.Text;  
  
namespace ConsoleApplication1  
{  
    public class Imprimante  
    { // champs  
        private static uint nombreImprimantes;  
        private string imprimanteNom;  
        private int numeroSalle;  
  
        // accesseurs(get/set)  
        public string ImprimanteNom  
        // notez la majuscule  
        // qui différencie le champ  
        // de la propriété  
        { get { return (imprimanteNom); }  
          set { imprimanteNom = value; }  
        }  
        public int NumeroSalle  
        {get { return (numeroSalle); }  
         set { numeroSalle = value; }  
        }  
  
        //Constructeur  
        public Imprimante(string name, int num)  
        {  
            nombreImprimantes++; // incremente nbre  
            imprimanteNom = name; // mise a jour  
            numeroSalle = num; // mise a jour  
        }  
  
        // methode statique  
        public static uint  
        AfficheTotalImprimanteGEII()  
  
        {  
            Console.WriteLine("le nombre total est:"  
+ nombreImprimantes);  
            return (nombreImprimantes);  
        }  
  
        // surcharge de la methode ToString pour ce  
        decrire public override string ToString()  
        {  
            Console.WriteLine("imprimante {0}  
localisée en salle {1}", imprimanteNom, numeroSalle);  
            return base.ToString();  
        }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Imprimante brother = new  
            Imprimante("brother", 215);  
            Imprimante hp = new Imprimante("HP",  
            208);  
            Imprimante hp2 = new Imprimante("HP",  
            214);  
            brother.ToString();  
            hp.ToString();  
            Console.WriteLine("salle pour hp2: " +  
            hp2.NumeroSalle);  
            uint  
            totalAtelier=Imprimante.AfficheTotalImprimanteGEII();  
            Console.ReadKey();  
        }  
    }  
}
```

Résultat écran

```
imprimante brother localisée en salle 215  
imprimante HP localisée en salle 208  
salle pour hp2: 214  
le nombre total est:3
```

constructeur

déclaration de constructeur

Dans tout programme le développeur doit se préoccuper de l'initialisation des objets créés. Il peut comme dans les cas précédents incorporer des méthodes spécifiques. Cependant le langage C++ inclut une fonction membre spéciale appelée constructeur. Cette méthode, du même nom que la classe, est appelée automatiquement au moment de la création d'un objet.

De la même manière, au moment de la destruction de cet objet, une fonction spéciale du nom de la classe mais précédé d'un tilde ~, est invoqué.

```
using
System;

class Chat
déclaration de la classe
{
    // propriété automatique (private)
    // avec avec accesseur public
    //compatible C# 3
    public ushort age { get; set; }
    public string nom { get; set; }

    // constructeur
    public Chat(ushort initialAge)
    { age=initialAge;}

    // destructeur (finalyser)
    ~Chat()
    {Console.WriteLine("chat détruit");}

    // définition de la méthode Miauler
    // renvoie : void
}

// paramètres : Aucun
// rôle : imprime "Miaou" à l'écran
public void Miauler()
{ Console.WriteLine("Miaou.."); }

}

class Program
{
    static void Main(string[] args)
    { Chat Frisky=new Chat(5);
      // au moment de la création le
      // constructeur est //automatiquement appelé
      Frisky.Miauler();
      Console.WriteLine( "Frisky est un
      chat qui a " +Frisky.age+" ans");
      Frisky.age=7;
      Console.WriteLine( "Maintenant
      Frisky a " +Frisky.age+" ans"); ;
      Console.ReadKey();
    }
}
```

Surcharge de constructeur

Si un constructeur avec argument a été défini alors la déclaration d'objet non initialisé est impossible : dans l'exemple précédent : Chat Frisky ; générerait une erreur. Alors que : Chat Frisky(5); est autorisé. Solution : utiliser le mécanisme de surcharge de fonction. C'est à dire, écrire les différents constructeurs qui correspondent au différent cas de figure des initialisations souhaitées.

Ci-après un exemple de surcharge du constructeur Chat.

```
using System;

class Chat
déclaration de la classe
{
    // propriété automatique (private)
    // avec avec accesseur public
    //compatible C# 3
    public ushort age { get; set; }
    public string nom { get; set; }

    // constructeurs
    public Chat(ushort initialAge)
    { age=initialAge;}

    public Chat()
    { age = 0; }

    public Chat(ushort initialAge,string
nom)
    { age = initialAge; this.nom = nom; }

    // destructeur (finalyser)
    ~Chat()
    {Console.WriteLine("chat détruit");}

}

// définition de la méthode Miauler
// renvoie : void
// paramètres : Aucun
// rôle : imprime "Miaou" à l'écran
public void Miauler()
{ Console.WriteLine("Miaou.."); }

}

class Program
{
    static void Main(string[] args)
    {
        Chat [] TabChat=new Chat[3];
        TabChat[0] = new Chat();
        TabChat[1] = new Chat(10);
        TabChat[2] = new Chat(10,"frisky");
        // au moment de la création le
        // structureur estautomatiquement appelé
        TabChat[2].Miauler();
        Console.WriteLine("Ce chat est un chat qui a " +
        TabChat[2].age + " ans");
        TabChat[2].age = 7;
        Console.WriteLine("CE chat a " + TabChat[2].age + "
ans");
        Console.WriteLine("Il s'appelle " + TabChat[1].nom);
        Console.ReadKey();
    }
}
```

TRAVAIL PERSONNEL

1- Avant de tester ce programme pronostiquer l'affichage?(en justifiant!!)Le saisir et vérifier **en pas à pas**.

```
namespace ConsoleApplication1
{
    class MonInt
    {
        public int val { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MonInt refUnInt=new MonInt();
            int x = 3;
            refUnInt.val=10;
            fct(x,refUnInt);
            Console.WriteLine( "x: {0} val:
{1}",x,refUnInt.val);
            Console.ReadLine();
        }
    }
    static void fct(int y,MonInt test)
    {
        test.val=20;
        y = 30;
    }
}
```

2- Modifier (en faisant un nouveau projet) le programme précédent de telle manière à réaliser un passage par référence sur la variable x initialisée. Conclusion. Refaire maintenant sur x non initialisée.Apporter les modifications pour que cela fonctionne.

3- Soit le programme suivant

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] tab = .....
            fct(tab);
            foreach (.....)
                Console.WriteLine(...);
            Console.ReadKey();
        }
        static void fct(int [] tab2)
        {
            //ajouter 1 à chaque élément
        }
    }
}
```

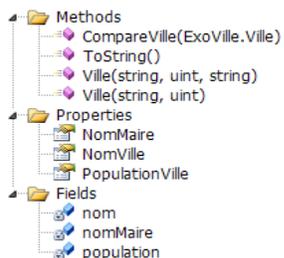
Avant saisie:

- Le contenu du tableau sera-t-il modifié?
- Peut-on supprimer le tableau (tab2=null) dans la fonction?
- Peut-on faire 'repointer' tab vers un nouveau tableau qui aurait été crée dans la fonction?
- Peut-on redimensionner tab dans la fonction ? `Array.Resize(ref tab2, 3);`

Vérifier vos réponses en faisant du pas à pas.

4- Même travail que précédemment mais le passage se fera par référence?

5-Créer une classe ville



ExoVille est l'espace de nommage (namespace)

CompareVille compare 2 villes (nom et population identique)

- ✓ Un argument de type Ville en entrée
- ✓ Un résultat en sortie de type bool (booléen)

A faire en TP: ToString est la réécriture de la méthode ToString (voir exemple dans ce chapitre). Cette fonction renvoie un string . Vous utiliserez `string.format` pour réaliser dans le "main" un affichage du type:

Ville Toulon , population 128000, maire Falco

Le programme principal sera simple et mettra en œuvre la classe Ville

- ✓ Création des objets "à la main" (pas de `console.read`) à l'aide des différents constructeurs
- ✓ Affichage des différents champs
- ✓ Tester CompareVille

6- faire des recherches sur internet pour comprendre les notions suivantes (important pour votre TP): multitâches, processus, thread, ordonnancement (

TP

1. Reprendre la classe Ville du travail personnel

1-1 Ecrire ToString. Tester avec une ville (`public override string ToString() { // code ici = }`)

1-2 Créer 'à la main' un tableau de 5 villes. Réaliser l'affichage à l'aide d'une boucle.

1-3 Rajouter une méthode statique CompareVille avec 2 arguments ville1 et ville2. Tester.

1-4 Rajouter la méthode d'instance

```
public void AdditionnePopulation(Ville nomVille2, ????? uint resultat)
```

Cette méthode doit pouvoir s'utiliser avec une variable non initialisée . Par quoi faut-il remplacer ????? pour que le résultat soit l'argument de sortie. Tester.

2. Envoi de mail

2.1. Installez ArgoSoft

Paramétrer le DNS local : par exemple @geii.fr et @iut-geii.fr

Créer des utilisateurs

2.2. examiner sur msdn en ligne la classe SmtpClient et MailMessage

Ecrire le programme le plus simple possible (pas de méthodes asynchrones!) pour envoyer un mail

Lancer outlook express (Démarrer>exécuter>msimn.exe). Paramétrer les comptes (127.0.0.1) et

vérifier les mails reçus

En utilisant la classe Attachment joindre en pièce jointe un fichier image (bmp ou jpg)

4. Première application graphique :IHM d'envoi de mail

Pour ce faire vous utiliserez les classes `TextBox`, `Button` et `ComboBox`.

Dans un premier temps tous les objets seront placés de manière statique sur le formulaire avec la boîte à outils de Visual Studio.

Pour récupérer la valeur d'une comboBox vous pouvez faire:

```
string to = ComboBoxDestinataire.Text; avec ComboBoxDestinataire du type comboBox
```

où

```
string to = (string) ComboBoxDestinataire.SelectedItem
```

Dans un deuxième temps les deux comboBox seront placées (créées) de manière dynamique au moment de l'initialisation du formulaire (ie appel du constructeur):

```
public partial class Form1 : Form
{
    //user
    private ComboBox ComboBoxDestinataire;
    private ComboBox ComboBoxCopieCC;
    private string[] annuaire;

    public Form1()
    {
        InitializeComponent();
        annuaire = new string[] { "", "oscar@geii.fr", "test@geii.fr", "amel@iut-geii.fr" };
        ComboBoxDestinataire = new ComboBox();
        ComboBoxDestinataire.Location = new Point(this.textBoxMessage.Location.X, this.labelTo.Location.Y);
        ComboBoxDestinataire.Items.AddRange(annuaire);
        this.Controls.Add(ComboBoxDestinataire);
    }
}
```

CLASSES COURANTES UTILISEES

Les cultures

Nous avons vu qu'il était possible d'utiliser des formateurs standards pour spécifier le résultat d'un l'affichage Le C# fournit 3 classes permettant d'obtenir un contrôle plus personnalisé des données manipulées:

- CultureInfo
- NumberFormatInfo
- DateTimeFormatInfo

Classe CultureInfo

La classe CultureInfo (`using System.Globalization;`) rend des informations spécifiques à une culture, par exemple la langue, la sous-langue, le pays et/ou la région ou le calendrier associé à la culture, ainsi que les conventions applicables à cette dernière.

Voici des exemples concrets de différences rencontrées en fonction de la nationalité de l'utilisateur:

- les Anglo-Saxons représentent les dates en intervertissant les jours et mois: MM/JJ/AA
- certains pays privilégient le point . au lieu de la , comme séparateur de la partie fractionnaire : 13.5 ou 13,5
- pour manipuler la monnaie le symbole représentant la devise varie en fonction des pays.

Culture localisé dans l'application: classe CultureInfo

```
// pb d'affichage de € en mode console
// sol: passer en Lucida console
// changer l'encodage de la console en UTF8 (représentation Unicode)
Console.OutputEncoding = Encoding.UTF8;
////////////////////////////////////
CultureInfo culture_courante = CultureInfo.CurrentUICulture; // methode statique: retourne culture courante
Console.WriteLine("culture courante "+ culture_courante.DisplayName);
decimal prix=12.3M;
CultureInfo ci_US = new CultureInfo("en-US"); // nvl objet culture USA
string text_prix_US = prix.ToString("c",ci_US);
Console.WriteLine(ci_US.DisplayName + ":");
Console.WriteLine(text_prix_US);
CultureInfo ci_Es = new CultureInfo("es-Es", false);//nvl objet culture Espagnole
string text_prix_Es = string.Format("{0:c}", ci_Es);// variante au .ToString
Console.WriteLine(ci_Es.DisplayName+":");
Console.WriteLine(text_prix_Es);
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-GB"); //
string text_prix = prix.ToString("c");
Console.WriteLine(text_prix);
Console.ReadKey();
```

culture courante Français (France) Anglais (États-Unis): \$12.30 Espagnol (Espagne): es-ES £12.30
--

Culture globalisée pour l'application

```
using System.Globalization;
using System.Threading;
.....
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
double d = 1.23;
string s = d.ToString(); // s contient 1.23
s = d.ToString("C"); // s contient $1.23
```

Classe NumberFormatInfo

Cette classe permet de personnaliser le formatage des nombres

Exemple 1: Accepte le . ou , comme séparateur de partie décimale

```
Console.WriteLine("PrixHTsvp:");
string chainePrixHT=Console.ReadLine();
if(chainePrixHT.Contains("."))
{
Var f=new System.Globalization.NumberFormatInfo();
f.NumberDecimalSeparator=".";
prixHT=double.Parse(chainePrixHT,f);
}
else
prixHT=double.Parse(chainePrixHT);
```

Exemple2 : Changement du symbole de devise

```
.....
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine (3.ToString ("C", f)); // $$
3.00
.....
```

Classe DateTimeFormatInfo

Cette classe permet d'obtenir des formatages spécifiques des dates (en plus des formateurs prédéfinis par le C# avec la méthode .ToString – voir classes dates plus loin dans le poly)

Exemple 1

```
// creation par exemple d'un nvl DateTimeFormatInfo à partir de l'original
// ne pas faire DateTimeFormatInfo mydateformat =ciFr.DateTimeFormat car juste nouvelle reference sur le meme
objet!
DateTimeFormatInfo mydateformat = (DateTimeFormatInfo)ciFr.DateTimeFormat.Clone(); // creation d'un nvl
objet!
string[] jourSemaine = { "L", "Mar", "Mer", "Jeud", "V", "S", "D" };
mydateformat.AbbreviatedDayNames = jourSemaine;
string[] jourSemaineOrigine=ciFr.DateTimeFormat.AbbreviatedDayNames;
foreach(string jour in jourSemaineOrigine) Console.Write(jour );
Console.WriteLine("temps actuel:" + datenow.ToString("ddd"));
Console.WriteLine("temps actuel:" + datenow.ToString("ddd", mydateformat));
```

temps actuel	:13/10/2010 10:59:01
temps actuel culture modifiée	:13-10-2010 10:59:01

Exemple 2

```
CultureInfo ciFr=new CultureInfo("fr-FR");
// ne pas faire DateTimeFormatInfo mydateformat =ciFr.DateTimeFormat car juste nouvelle reference sur le meme
objet!
DateTimeFormatInfo mydateformat = (DateTimeFormatInfo)ciFr.DateTimeFormat.Clone(); // creation d'un nvl
objet!
string[] jourSemaine = { "D","L", "Ma", "Me", "Jeud", "V", "S" };
mydateformat.AbbreviatedDayNames = jourSemaine;
string[] jourSemaineOrigine=ciFr.DateTimeFormat.AbbreviatedDayNames;
Console.WriteLine("Affichage jour d'origine abrégé");
foreach(string jour in jourSemaineOrigine) Console.Write(jour+" ");
DateTime datenow = DateTime.Now;
Console.WriteLine("\ntemps actuel:" + datenow.ToString("ddd"));
Console.WriteLine("temps actuel:" + datenow.ToString("ddd", mydateformat));
Console.ReadKey();
```

Affichage jour d'origine abrégé
dim. lun. mar. mer. jeu. ven.
sam.
temps actuel:mer.
temps actuel:Me

Le Temps et les dates

Type *TimeSpan*

Un objet **TimeSpan** représente un intervalle de temps, ou une durée, mesuré en nombre positif ou négatif de jours, heures, minutes, secondes et fractions de seconde

La valeur d'un objet **TimeSpan** est le nombre de graduations équivalant à l'intervalle de temps représenté.

Une graduation est égale à 100 nanosecondes.

ATTENTION: TimeSpan est une structure, donc du type valeur!

Les constructeurs

TimeSpan (Int64)	Initialise un nouveau TimeSpan avec le nombre de graduations spécifié. Pris en charge par le .NET Compact Framework.
TimeSpan (Int32, Int32, Int32)	Initialise un nouveau TimeSpan par rapport à un nombre d'heures, de minutes et de secondes spécifié. Pris en charge par le .NET Compact Framework.
TimeSpan (Int32, Int32, Int32, Int32)	Initialise un nouveau TimeSpan par rapport à un nombre de jours, d'heures, de minutes et de secondes spécifié. Pris en charge par le .NET Compact Framework.
TimeSpan (Int32, Int32, Int32, Int32, Int32)	Initialise un nouveau TimeSpan par rapport à un nombre de jours, d'heures, de minutes, de secondes et de millisecondes spécifié. Pris en charge par le .NET Compact Framework.

Les Propriétés

Days	Obtient le nombre de jours entiers représentés par la structure TimeSpan actuelle.
Hours	Obtient le nombre d'heures entières représentées par la structure TimeSpan actuelle.
Milliseconds	Obtient le nombre de millisecondes entières représentées par la structure TimeSpan actuelle.
Minutes	Obtient le nombre de minutes entières représentées par la structure TimeSpan en cours.
Seconds	Obtient le nombre de secondes entières représentées par la structure TimeSpan en cours.
Ticks	Obtient le nombre de graduations représentant la valeur de la structure TimeSpan en cours.
TotalDays	Obtient la valeur de la structure TimeSpan exprimée en jours entiers et fractionnaires.
TotalHours	Obtient la valeur de la structure TimeSpan exprimée en heures entières et fractionnaires.
TotalMilliseconds	Obtient la valeur de la structure TimeSpan exprimée en millisecondes entières et fractionnaires.
TotalMinutes	Obtient la valeur de la structure TimeSpan exprimée en minutes entières et fractionnaires.
TotalSeconds	Obtient la valeur de la structure TimeSpan exprimée en secondes entières et fractionnaires.

Des Méthodes

Static	FromDays	Retourne un TimeSpan représentant un nombre de jours spécifié, où la spécification est précise à la milliseconde près.
S	FromHours	Retourne un TimeSpan représentant un nombre d'heures spécifié, où la spécification est précise à la milliseconde près.
S	FromMilliseconds	Retourne un TimeSpan représentant un nombre spécifié de millisecondes.
S	FromMinutes	Retourne un TimeSpan représentant un nombre de minutes spécifié, où la spécification est précise à la milliseconde près.
S	FromSeconds	Retourne un TimeSpan représentant un nombre de secondes spécifié, où la spécification est précise à la milliseconde près.
	Add	Ajoute le TimeSpan spécifié à cette instance.
	Subtract	Soustrait le TimeSpan spécifié de cette instance.
	TryParse	Construit un nouvel objet TimeSpan à partir d'un intervalle de temps spécifié dans une chaîne de caractères. Les paramètres spécifient l'intervalle de temps et la variable dans laquelle le nouvel objet TimeSpan est retourné.

Exemples

```
Console.WriteLine (new TimeSpan (2, 30, 0)); // 02:30:00
Console.WriteLine (TimeSpan.FromHours (2.5)); // 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5)); // -02:30:00
TimeSpan nearlyTenDays =TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1); // 9.23:59:59
Console.WriteLine (nearlyTenDays.Days); // 9
Console.WriteLine (nearlyTenDays.Hours); // 23
Console.WriteLine (nearlyTenDays.Minutes); // 59
Console.WriteLine (nearlyTenDays.TotalDays); // 9.99998842592593
Console.WriteLine (nearlyTenDays.TotalHours); // 239.999722222222
```

Type **DateTime**

Le type valeur **DateTime** représente des dates et des heures. Les valeurs de date sont mesurées en unités de 100 nanosecondes, appelées graduations, et une date spécifique correspond au nombre de graduations écoulées à compter de 12:00 (minuit), le 1er janvier de l'année 0001 après J.C. de l'ère commune dans le calendrier [GregorianCalendar](#). Par exemple, une valeur de graduations égale à 312413760000000000L représente la date du vendredi 1er janvier 0100 12:00:00 (minuit).

Une valeur **DateTime** est toujours exprimée dans le contexte d'un calendrier explicite ou par défaut.

Avec le type **DateTime** il est possible de représenter une date dans un repère:

- locale: liée à l'horloge de votre ordinateur (horloge windows)
- UTC (Coordinated Universal Time): heure universelle (anciennement Greenweech)
 - o Le temps universel coordonné est l'heure mesurée à la longitude zéro, le point de départ de l'UTC. L'heure d'été ne s'applique pas à l'UTC.
- Non spécifié

Constructeurs

DateTime (Int64, DateTimeKind)	Initialise une nouvelle instance de la structure DateTime avec un nombre spécifié de graduations et une heure UTC ou locale. Pris en charge par le .NET Compact Framework.
DateTime (Int32, Int32, Int32)	Initialise une nouvelle instance de la structure DateTime avec l'année, le mois et le jour spécifiés. Pris en charge par le .NET Compact Framework.
DateTime (Int32, Int32, Int32, Calendar)	Initialise une nouvelle instance de la structure DateTime avec l'année, le mois et le jour spécifiés pour le calendrier spécifié. Pris en charge par le .NET Compact Framework.
DateTime (Int32, Int32, Int32, Int32, Int32, Int32)	Initialise une nouvelle instance de la structure DateTime avec l'année, le mois, le jour, la minute et la seconde spécifiés. Pris en charge par le .NET Compact Framework.
DateTime (Int32, Int32, Int32, Int32, Int32, Int32, Calendar)	Initialise une nouvelle instance de la structure DateTime avec l'année, le mois, le jour, la minute et la seconde spécifiés pour le calendrier spécifié. Pris en charge par le .NET Compact Framework.
DateTime (Int32, Int32, Int32, Int32, Int32, Int32, DateTimeKind)	Initialise une nouvelle instance de la structure DateTime avec l'année, le mois, le jour, l'heure, la minute, la seconde, et l'heure UTC ou locale. Pris en charge par le .NET Compact Framework.

Exemple:

```
DateTime dt1 = new DateTime(2008, 10, 2, 23, 3, 59, 10, DateTimeKind.Utc);
DateTime dt2 = new DateTime(2008, 10, 2, new System.Globalization.JapaneseCalendar());
Console.WriteLine("date : {0}", dt1);
Console.WriteLine("date : {0}", dt2);
Console.WriteLine(dt2.Kind);
```

Ci-après quelques méthodes (liste non exhaustive – voir msdn en ligne pour compléments d'infos).

	Nom	Description
	Add	Ajoute la valeur du TimeSpan spécifié à la valeur de cette instance.
	AddDays	Ajoute le nombre de jours spécifié à la valeur de cette instance.
	AddHours	Ajoute le nombre d'heures spécifié à la valeur de cette instance.
	AddMinutes	Ajoute le nombre de minutes spécifié à la valeur de cette instance.
	AddMonths	Ajoute le nombre de mois spécifié à la valeur de cette instance.
	AddYears	Ajoute le nombre d'années spécifié à la valeur de cette instance.
S	Compare	Compare deux instances de DateTime et retourne une indication de leurs valeurs relatives.
	DaysInMonth	Retourne le nombre de jours compris dans le mois et l'année spécifiés.
	Equals	Surchargé. Retourne une valeur indiquant si deux objets DateTime , ou si une instance DateTime et un autre objet ou DateTime sont égaux.
S	op_Addition	Ajoute un intervalle de temps spécifié à une date et une heure spécifiées, générant une nouvelle date et heure.
S	op_Equality	Détermine si deux instances spécifiées de DateTime sont égales.
S	op_GreaterThan	Détermine si un DateTime spécifié est supérieur à un autre DateTime spécifié.
S	Parse	Surchargé. Convertit la représentation sous forme de chaîne spécifiée d'une date et d'une heure en DateTime équivalent.
	Subtract	Surchargé. Soustrait l'heure ou la durée spécifiée de cette instance.
	ToString	Surchargé. Substitué. Convertit la valeur de cette instance en sa représentation sous forme de chaîne équivalente.
	ToUniversalTime	Convertit la valeur de l'objet DateTime en cours en Temps universel coordonné (UTC).
S	TryParse	Surchargé. Convertit la représentation sous forme de chaîne spécifiée d'une date et d'une heure en DateTime équivalent.

Exemple

```
DateTime dt3 = new DateTime(2008, 10, 2);
DateTime dt4 = new DateTime(2008, 10, 4);
TimeSpan ts = TimeSpan.FromDays(2);
Console.WriteLine(dt3 == dt4);
Console.WriteLine(dt3 > dt4);
Console.WriteLine(DateTime.Compare(dt3, dt4));
dt3=dt3.Add(ts);
Console.WriteLine(DateTime.Compare(dt3, dt4));
dt3 = dt3 + ts;
Console.WriteLine(DateTime.Compare(dt3, dt4));
```

False
False
-1
0
1

Propriétés

Nom	Description
Date	Obtient le composant "date" de cette instance.
Day	Obtient le jour du mois représenté par cette instance.
DayOfWeek	Obtient le jour de semaine représenté par cette instance.
DayOfYear	Obtient le jour de l'année représenté par cette instance.
Hour	Obtient le composant "heure" de la date représentée par cette instance.
Kind	Obtient une valeur qui indique si l'heure représentée par cette instance se base sur l'heure locale, l'heure UTC, ou aucune des deux.
Millisecond	Obtient le composant "millisecondes" de la date représentée par cette instance.
Minute	Obtient le composant "minutes" de la date représentée par cette instance.
Month	Obtient le composant "mois" de la date représentée par cette instance.
Now	Obtient un objet DateTime qui a pour valeur la date et l'heure actuelles sur cet ordinateur, exprimées en temps local.
Second	Obtient le composant "secondes" de la date représentée par cette instance.
Ticks	Obtient le nombre de graduations représentant la date et l'heure de cette instance.
TimeOfDay	Obtient l'heure de cette instance.
Today	Obtient la date actuelle.
UtcNow	Obtient un objet DateTime qui a pour valeur la date et l'heure actuelles sur cet ordinateur, exprimées en temps UTC.
Year	Obtient le composant "année" de la date représentée par cette instance.

Affichage et formatage des dates

Les formateurs permettent de personnaliser des dates pour l'affichage ou la conversion de date. Ils sont à utiliser par exemple avec les méthodes:

- ✓ Console.WriteLine
- ✓ objets DateTime.ToString
- ✓ String.Format
- ✓ String.Parse

Formats standards

dépendant de la culture courante

Format string	Meaning	Sample output
d	Short date	01/02/2000
D	Long date	Sunday, 02 January 2000
t	Short time	17:18
T	Long time	17:18:19
f	Long date + short time	Sunday, 02 January 2000 17:18
F	Long date + long time	Sunday, 02 January 2000 17:18:19
g	Short date + short time	01/02/2000 17:18
G (default)	Short date + long time	01/02/2000 17:18:19
m, M	Month and day	January 02
y, Y	Year and month	2000 January

Formats standards INSENSIBLE à la culture

Format string	Meaning	Sample output	Notes
o	Round-trippable	2000-01-02T17:18:19.0000000	Will append time zone information unless DateTimeKind is Unspecified
r, R	RFC 1123 standard	Sun, 02 Jan 2000 17:18:19 GMT	You must explicitly convert to UTC with DateTime.ToUniversalTime
s	Sortable; ISO 8601	2000-01-02T17:18:19	Compatible with text-based sorting
u	"Universal" Sortable	2000-01-02 17:18:19Z	Similar to above; must explicitly convert to UTC
U	UTC	Sunday, 02 January 2000 17:18:19	Long date + short time, converted to UTC

Exemple:

```
DateTime dt5 = new DateTime(2008, 10, 2, 23, 3, 59, 10);
Console.WriteLine("affichage format court {0:d}", dt5);
string str_convert = string.Format("la meme au format long {0:D}", dt5);
Console.WriteLine(str_convert);
Console.WriteLine(dt5.ToString("dddd, le d MMMM yyyy"));
Console.WriteLine(dt5.ToString("ce dd/MMM à HH:mm"));
Console.WriteLine(dt5.ToString("à HH heures mm minutes"));
Console.WriteLine(dt5.ToString("à HH 'heures' mm 'minutes'"));
string str = "2/11/09";
DateTime dt7;
if (DateTime.TryParse(str, out dt7) == false)
    Console.WriteLine("echec du PArse nouvelle saisie");
else
    Console.WriteLine("date convertie ds la culture {0} {1:d}",
        System.Globalization.CultureInfo.CurrentUICulture, dt7);
    DateTime dt8;
dt8=DateTime.Parse(str, System.Globalization.CultureInfo.InvariantCulture)
Console.WriteLine("dans InvariantCulture mois et jour sont inversé {0:d}", dt8);
```

affichage format court 02/10/2008
la meme au format long jeudi 2 octobre 2008
jeudi, le 2 octobre 2008
ce 02/oct. à 23:03
à 23 11heure59 03 3inue59
à 23 heures 03 minutes
date convertie ds la culture fr-FR 02/11/2009
dans InvariantCulture mois et jour sont inversé 11/02/2009

REMARQUE: Des erreurs d'interprétation sont possibles lorsque les dates sont données par une tierce partie (exemple: fichier texte). En effet une ambiguïté sur l'ordre du mois et du jour existe. C'est la raison

pour laquelle il est préférable d'exporter les dates en précisant l'année en premier. Des formateurs non sensibles à la culture existent:

Préférez le formateur 'o' qui convertit automatiquement en UTC

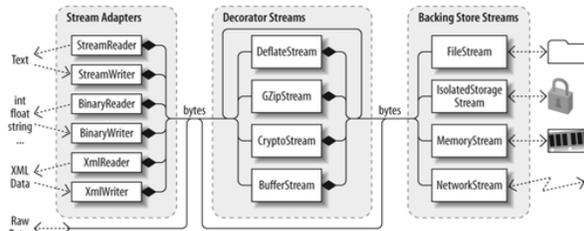
```
DateTime dt9=new DateTime(2009,5,3,13,10,53);
// J=5/M=3/A=2009 13h10mn53s
Console.WriteLine("culture dependant {0:d}", dt9);
Console.WriteLine("insensible culture {0:o}", dt9);
```

Pour relire:

```
DateTime dt1 = DateTime.ParseExact(s, "o", null); ou
DateTime dt2 = DateTime.Parse(s); // "o" utilisé par défaut
```

Les flux

Stream est la classe de base abstraite de tous les flux. Un flux est une abstraction d'une séquence d'octets, telle qu'un fichier, un périphérique d'entrée/sortie, un canal de communication à processus interne, ou un socket TCP/IP. La classe **Stream** et ses classes dérivées donnent une vue générique de ces différents types d'entrées et de sorties, isolant le programmeur des détails spécifiques au système d'exploitation et aux périphériques sous-jacents.



Classe de "stockage physique"
Elle dérive de la classe abstraite Stream et implémente les fonctionnalités de bases de lecture et d'écriture vers le support physique ciblé

Classes "adaptateur"

Classe d'encodage permettant une manipulation des données par l'utilisateur sous une forme plus évoluée.
Ex:
octet -> string
Octet -> xml

Classes "Décorateur"

Classe d'encodage intermédiaire.
Ex: cryptage des données

Schéma de principe d'utilisation

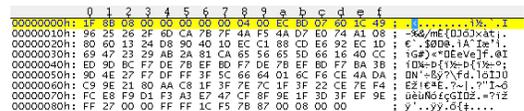
Ecriture simple dans un fichier

```
using (FileStream fs = new FileStream("test.txt", FileMode.Create))
{
    byte [] tab = new byte[] {0x49, 0x55, 0x54}; // "IUT" écrit
    fs.Write(tab, 0, tab.Length);
}
```



Ecriture simple dans un fichier avec

```
using (FileStream fs2 = new FileStream("test2.gztxt", FileMode.Create))
{
    using (GZipStream compress_fs = new GZipStream(fs2, CompressionMode.Compress))
    {
        byte[] tab2 = new byte[2048];
        for (uint i=0; i<tab2.Length; i++) tab2[i] = 0x49;
        compress_fs.Write(tab2, 0, tab2.Length);
    }
}
```



On constate que la taille du fichier est bien inférieure à 2Ko

La classe FileStream

Classe qui implémente pour les fichiers les méthodes générales de la classe générique stream (concepts héritage et interface vus plus loin) tels que :

```
public override int Read(byte[] array, int offset, int count) ou
public override void Write(byte[] array, int offset, int count)
```

Exemple

```
using System;
using System.IO;
using System.Text;

namespace test
{
    class Program
    {
        static void Main(string[] args)
        {
            string path=@"G:\Temp\";
            string namefile="capteur.txt";
            FileStream fs = new FileStream(path + namefile, FileMode.Create);
            byte[] ByteArray=new byte[20];
            fs.WriteByte(0xFF);
            for (byte i=0;i< 20; i++) ByteArray[i]=i;
            // pas de prbleme de conversion car ByteArray
            // tableau d'octet
            fs.Write(ByteArray, 0, ByteArray.Length);
            string chaine = "je veux etre enregistre";
            // Convert the string into a byte[]
            Encoding unicode = Encoding.Unicode; // choix codage
            byte[] unicodechaine = unicode.GetBytes(chaine);
            fs.Write(unicodechaine, 0, unicode.GetByteCount(chaine));
            string chaine2 = "je veux etre enregistre en utf8";
            fs.Write(Encoding.UTF8.GetBytes(chaine2), 0, Encoding.UTF8.GetByteCount(chaine2));
            // conversion double vers string puis ASCII
            double val = 10.536;
            String strNumber = System.Convert.ToString(val);
            fs.Write(Encoding.ASCII.GetBytes(strNumber), 0,Encoding.ASCII.GetByteCount(strNumber));
            fs.Close();
        }
    }
}
```

```
00000000h: FF 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E ; .....
00000010h: 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E ; .....j.e .v.e.u
00000020h: 00 78 00 20 00 65 00 74 00 72 00 65 00 20 00 65 ; .x .e.t.r.e .e
00000030h: 00 6E 00 72 00 65 00 67 00 69 00 73 00 74 00 72 ; .n.r.e.g.i.s.t.r
00000040h: 00 65 00 6A 65 20 76 65 75 78 20 65 74 72 65 20 ; .e.je veux etre
00000050h: 65 6E 72 65 67 69 73 74 72 65 20 65 6E 20 75 74 ; enregistre en ut
00000060h: 66 38 ; f8
```

Commentaires: L'écriture de chaînes de caractères ou de nombres nécessite des conversions diverses. Afin de soulager le concepteur de programmes des classes dérivées sont proposés pour traiter les fichiers textes et binaire.

La classe NetworkStream

La classe **NetworkStream** fournit les méthodes pour l'envoi et la réception de données via des sockets Stream en mode blocage.

Tous les constructeurs requièrent un objet de type socket connecté. Il existe plusieurs méthodes:

- Manipulation de la classe socket et des méthodes associés (méthode plus flexible mais aussi plus complexe)
- Manipulation des classes TcpListener (côté serveur) et TcpClient (côté Client) et de la méthode GetStream() qui renvoie un socket connecté!

COTE CLIENT

```
TcpClient monTcpClient = new TcpClient("127.0.0.1", 10000);
using (NetworkStream ns = monTcpClient.GetStream())
{
    string chaine = "ma requete";
    Encoding encode = Encoding.UTF8;
    byte[] buffer = encode.GetBytes(chaine);
    ns.Write(buffer, 0, buffer.Length);
    Console.WriteLine("envoi de la requete");
    byte[] response = new byte[100];
    String string_response = String.Empty;
    Console.WriteLine("attente response");
    ns.Read(response, 0, response.Length);
    string_response = encode.GetString(response);
    Console.WriteLine(string_response);
    Console.WriteLine("fin");
    Console.ReadKey();
}
```

COTE SERVEUR

```
IPAddress ipAddress = IPAddress.Parse("127.0.0.1");
TcpListener server = new TcpListener(ipAddress, 10000);
server.Start();
Console.WriteLine("\n Attente connection... ");

TcpClient client = server.AcceptTcpClient();//bloquant
using (NetworkStream ns = client.GetStream())
{
    Encoding encode = Encoding.UTF8;
    byte[] requete = new byte[100];
    ns.Read(requete, 0, requete.Length);
    string strRequete = encode.GetString(requete);
    Console.WriteLine("message du client" + strRequete);
    byte[] reponse = encode.GetBytes("ACK");
    ns.Write(reponse, 0, reponse.Length);
    Console.WriteLine("\n Fin serveur... ");
    Console.ReadKey();
}
```

Commentaires:

Techniquement les flux de type réseaux n'ont pas de fin. La condition nombre d'octet lu égale 0 signifie simplement que le buffer de réception (au niveau système) a été vidé par les lectures précédentes. Il n'est donc pas sur que l'intégralité du message ait été envoyé (exemple d'une collision sur le réseau).

Il est préférable d'utiliser les classes `BinaryWriter` et `BinaryReader` qui facilitent la manipulation des données. De plus, en utilisant ces classes, les données sont préfixées par leur longueur. Les méthodes de lecture connaissent le nombre d'octets qu'elles doivent lire (ce n'est pas le cas avec `StreamReader` qui est déconseillé pour les accès réseaux)

Adaptateur de type Texte

Les classes (adaptateur) `StreamReader` et `StreamWriter` permettent de manipuler facilement du texte au travers de flux, et notamment les fichiers textes .

Application à la lecture ou écriture de fichier

Utilisez `StreamReader` pour lire des lignes d'informations à partir d'un fichier texte standard.

Utilisez `StreamWriter` pour écrire des lignes d'informations à partir d'un fichier texte standard.

Les classes exposent en plus du constructeur `StreamWriter(Stream)` (voir exemple ci-dessous en lecture) un constructeur `StreamWriter(String)` qui ouvre en "sous main" un flux de type fichier ,permettant la manipulation directe de ce fichier.

Sauf spécification contraire, le codage par défaut de `StreamReader` et `StreamWriter` est UTF-8.

LECTURE

Exemple Flux combiné

```
using System;
using System.IO;

public class CompBuf
{
    private const string FILE_NAME = "MyFile.txt";

    public static void Main()
    {
        if (!File.Exists(FILE_NAME))
        {
            Console.WriteLine("{0} does not exist!", FILE_NAME);
            return;
        }
        FileStream fsIn = new FileStream(FILE_NAME, FileMode.Open,
            FileAccess.Read, FileShare.Read);
        // Create an instance of StreamReader that can read
        // characters from the FileStream.
        using (StreamReader sr = new StreamReader(fsIn))
        {
            string input;
            // While not at the end of the file, read lines from the file.
            while (sr.Peek() > -1)
            {
                input = sr.ReadLine();
                Console.WriteLine(input);
            }
        }
    }
}
```

ECRITURE

Ecriture allégé: ouverture directe du fichier
`StreamWriter("TestFile.txt")`

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        // Create an instance of StreamWriter to write text to a file.
        // The using statement also closes the StreamWriter.
        using (StreamWriter sw = new StreamWriter("TestFile.txt"))
        {
            // Add some text to the file.
            sw.WriteLine("This is the ");
            sw.WriteLine("header for the file.");
            sw.WriteLine("-----");

            // Arbitrary objects can also be written to the file.
            sw.WriteLine("The date is: ");
            sw.WriteLine(DateTime.Now);
        }
    }
}
```

Complément sur la manipulation des fichiers

Ci après sont présentés quelques types livrés dans l'espace de nommage System.IO pour la manipulation sur les fichiers, tels que copie, déplacement, renommage d'extension etc...

Classe	Description
Directory	Expose des méthodes statiques pour créer, déplacer et énumérer via des répertoires et sous-répertoires. Cette classe ne peut pas être héritée.
DirectoryInfo	Expose des méthodes d'instance pour créer, déplacer et énumérer des répertoires et sous-répertoires. Cette classe ne peut pas être héritée.
File	Fournit des méthodes statiques pour créer, copier, supprimer, déplacer et ouvrir des fichiers et facilite la création d'objets FileStream .
FileInfo	Fournit des méthodes d'instance pour créer, copier, supprimer, déplacer et ouvrir des fichiers et facilite la création d'objets FileStream . Cette classe ne peut pas être héritée.
Path	Exécute des opérations sur des instances de String qui contiennent des informations relatives au chemin d'accès d'un fichier ou d'un répertoire. Ces opérations sont exécutées différemment selon la plateforme.

EXEMPLE 1

```
using System;
using System.IO;
class Test
{ public static void Main()
  {
    string path = Path.GetTempFileName();
    FileInfo fi1 = new FileInfo(path);
    if (!fi1.Exists) {
      //Create a file to write to.
      using (StreamWriter sw = fi1.CreateText())
      {
        sw.WriteLine("Hello");
        sw.WriteLine("And");
        sw.WriteLine("Welcome");
      }
    }
    //Open the file to read from.
    using (StreamReader sr = fi1.OpenText())
    {
      string s = "";
      while ((s = sr.ReadLine()) != null)
      {
        Console.WriteLine(s);
      }
    }
    try
    {
      string path2 = Path.GetTempFileName();
      FileInfo fi2 = new FileInfo(path2);
      //Ensure that the target does not exist.
      fi2.Delete();
      //Copy the file.
      fi1.CopyTo(path2);
      Console.WriteLine("{0} was copied to {1}.", path, path2);
      //Delete the newly created file.
      fi2.Delete();
      Console.WriteLine("{0} was successfully deleted.", path2);
    }
    catch (Exception e)
    {
      Console.WriteLine("The process failed: {0}", e.ToString());
    }
  }
}
```

EXEMPLE 2

```
using System;
using System.IO;

namespace streamwriter
{
  class Program
  {
    public const ushort nbelt=5;

    static void Main(string[] args)
    {
      string path=@"G:\Temp\";
      string namefile="capteur.txt";

      if (!Directory.Exists (path))
        Directory.CreateDirectory (path);

      FileInfo fic=new FileInfo(path+namefile);
      ConsoleKeyInfo keypressed;
      StreamWriter fictxt=null;

      if (fic.Exists)
      {
        Console.WriteLine("LE fichier" + namefile + " existe. Souhaitez vous l'écraser?");
        keypressed = Console.ReadKey(true);
        if (keypressed.KeyChar=='O')
          fictxt=fic.CreateText();
        else
          fictxt = fic.AppendText();
      }

      fictxt.WriteLine("capteur no;echantillon no;valeur");
      for (int i=0;j< nbelt; i++)
      {
        float val;
        val = (float)Math.Sin((2*Math.PI/ nbelt)*i);
        string concat = "capeur1;" + i + ";" + val;
        fictxt.WriteLine(concat);
      }
      fictxt.Close();
    }
  }
}
```

Adapteur de type Binaire

Les classes BinaryWriter et BinaryReader permettent de manipuler les objets natifs C# au travers d'un flux.

Exemple en écriture

```
public class Fiche
{
    string m_nom, m_prenom;
    byte m_age;
    public string Nom
    {
        get { return (m_nom); }
        set { m_nom = value; }
    }
    public string Prenom
    {
        get { return (m_prenom); }
        set { m_prenom = value; }
    }
    public byte Age
    {
        get { return (m_age); }
        set { m_age = value; }
    }
}

public Fiche(string nom, string prenom, byte
age)
{ Nom = nom; Prenom = prenom; Age = age; }

public override string ToString()
{
    string strFiche = String.Format("Nom: {0}
Prenom: {1} agé(e) de {2} ans", Nom, Prenom, Age);
    return strFiche;
}
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Fiche[] tabFiche = new Fiche[2];
        tabFiche[0] = new Fiche("Amel", "Oscar",
20);
        tabFiche[1] = new Fiche("Copter", "Elie",
22);
        Console.WriteLine(tabFiche[0].ToString());

        using (FileStream fs =
File.Create("datafiche.bin"))
        {
            using (BinaryWriter fs_bin = new
BinaryWriter(fs))
            {
                foreach (Fiche unefiche in tabFiche)
                {
                    fs_bin.Write(unefiche.Nom);
                    fs_bin.Write(unefiche.Prenom);
                    fs_bin.Write(unefiche.Age);
                }
            }
        }
        Console.ReadKey();
    }
}
```

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	04	41	6D	65	6C	05	4F	73	63	61	72	14	06	43	6F	70	; .Amel.Oscar..Cop
00000010h:	74	65	72	04	45	6C	69	65	16								; ter.Elle.

Exemple de relecture du fichier

```
using (FileStream fs = File.OpenRead("datafiche.bin"))
{
    using (BinaryReader fs_binread = new BinaryReader(fs))
    {
        while (fs_binread.PeekChar() != -1) //lit car suivant sans déplacer
// l'index de lecture
        {
            string nomfiche=fs_binread.ReadString();
            string prenomfiche = fs_binread.ReadString();
            byte agefiche= fs_binread.ReadByte();
            Console.WriteLine(nomfiche + "/" + prenomfiche + "/" + agefiche.ToString());
            Console.ReadKey();
        }
    }
}
```

Commentaire:

PeekChar() est une façon de tester la fin de fichier lorsque l'on ne connaît pas la taille du fichier (ici de le nombre de fiche stockée)

La relecture d'un fichier binaire implique de connaître la structure interne de stockage des données : ici: string/string/byte.

Introduction aux collections

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

Classe Array

Elle sert de classe de base pour tous les tableaux du Common Language Runtime. Tous les tableaux que vous créez manipulés sont en fait de la classe Array.

Exemple

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Array_exemple1
{
    class Program
    {
        static void Main(string[] args)
        {
            /****** Array demo
            *****/
            byte[] tab = new byte[10];
            Random rnd1= new Random();
            rnd1.NextBytes(tab);
            foreach (byte ibyte in tab) Console.Write(ibyte + " ");
            if (((IList<byte>)tab).Contains(10)) Console.WriteLine("contient
la valeur 10");
            else Console.WriteLine("ne contient la valeur 10");
            Array.Sort(tab);
            foreach (byte ibyte in tab) Console.Write(ibyte + " ");
```

```
Console.WriteLine("\n nombre pair");
byte[] tab_pair = Array.FindAll(tab, Fct_Recherche_Pair);
foreach (byte ibyte in tab_pair) Console.Write(ibyte + " ");
Console.WriteLine("\n redimensionne tableau");
Array.Resize<byte>(ref tab, 20);
foreach (byte ibyte in tab) Console.Write(ibyte + " ");
```

```
Console.ReadKey();
}
```

```
private static bool Fct_Recherche_Pair(byte val)
{
    if ((val % 2) == 0)
        return (true);
    else
        return (false);
}
}
```

```
221 147 115 135 32 133 201 104 150 94 ne
contient la valeur 10
32 94 104 115 133 135 147 150 201 221
nombre pair
32 94 104 150
redimensionne tableau
32 94 104 115 133 135 147 150 201 221 0 0 0 0
0 0 0 0 0
```

Classe générique List<T>

'List' capable de stocker des références sur des objets de type <T>. Ce tableau de type liste est dynamique et sa taille peut varier à tout moment grâce aux méthodes associés à la classe List.

La création d'une liste nécessite au moment de la création la spécification du type de base des objets contenus. List est donc un conteneur générique (notion développée plus loin dans ce poly).

Exemple de création d'une liste d'entier: `List<int> maListeInt=new List<int>();`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Conteneur_exemple1
{
    class Program
    {
        static void Main(string[] args)
        {
            Random rnd2 = new Random();
            List<int> listTab = new List<int>(10); // capacité
de la liste:10 elts
            for (int i = 0; i < 10; i++)// ATTENTION: LA LISTE
EST VIDE ,CAPACITE différent de TAILLE
            // listTab[i] = 100; COMPILATEUR KO car la liste
est vide
            listTab.Add(rnd2.Next(1000)); // rajout nbre
random <1000
            foreach (int val in listTab) Console.Write(val + "
"); // parcours de la liste
            listTab[0] = 100; // acces à un elt EXISTANT
```

```
listTab.Insert(1, 101); // insertion elt 101 à
l'index 1
Console.WriteLine("\ncapacité de la liste
"+listTab.Capacity); //la capacité a été
redimensionné!
listTab.RemoveAt(2); // suppression elt à l'index
3
Console.WriteLine("\n Liste modifiée");
foreach (int val in listTab) Console.Write(val + "
"); // parcours de la liste
listTab.AddRange(new int[] { 10, 20, 30 }); //
rajoute une collection d'objet
Console.WriteLine("\n Liste modifiée 2");
foreach (int val in listTab) Console.Write(val + "
"); // parcours de la liste
Console.WriteLine("\nLemax est"+listTab.Max());
Console.WriteLine("\nLa moyenne est"+
listTab.Average());
Console.ReadKey();
}
}
```

Les dictionnaires

Un dictionnaire est une collection d'objet de type clé/ Valeur. Chaque clé, qui doit être unique, sert à référencer l'objet dans le dictionnaire. Les recherches peuvent se faire suivant la clé (le plus rapide) ou suivant le contenu de l'objet.

Différentes classes implémentent ce concept de dictionnaire suivant:

- Que les objets puissent être indexable (concept tableau)
- Que La collection soit ordonnée ou non
- Que La classe soit générique ou non
- Les performances souhaitées

Type	Internal structure	Re-trieve by index?	Memory overhead (avg. bytes per item)	Speed: random insertion	Speed: sequential insertion	Speed: re-trieval by key
Unsorted						
Dictionary <K,V>	Hashtable	No	22	30	30	20
Hashtable	Hashtable	No	38	50	50	30
LinkedListDictionary	LinkedList	No	36	50,000	50,000	50,000
OrderedDictionary	Hashtable + array	Yes	59	70	70	40
Sorted						
SortedDictionary <K,V>	Red/black tree	No	20	130	100	120
SortedList <K,V>	2xArray	Yes	2	3,300	30	40
SortedList	2xArray	Yes	27	4,500	100	180

Exemple: création et méthodes de base associés aux dictionnaires

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Dictionnaire_Simple1
{
    public class Etudiant
    {
        public string Nom {get;set;}
        public string Prenom {get;set;}
        public UInt16 Age {get;set;}
        public UInt32 CodeApogee {get;set;}

        public Etudiant(string nom,string
        prenom,UInt16age, UInt32 codeApogee)
        {Nom=nom;
        Prenom=prenom;Age=age;CodeApogee=codeApogee;}

        public override string ToString()
        {return (CodeApogee+" "+Nom+" "+Prenom+" age de
        "+Age.ToString()+" ans");}
    }

    class Program
    {
        static void Main(string[] args)
        { // creation dictionnaire
        Dictionary<UInt32,Etudiant> dicLPSE=
        new Dictionary<UInt32,Etudiant> ();
        // ajout des entrees
        dicLPSE.Add(100213,new
        Etudiant("Oscar","Amel",21,100213));
        dicLPSE.Add(100214,new
        Etudiant("Copter","Elie",20,100214));
        // acces d'un elt (affichage necessite surcharge
        de ToString)
        Console.WriteLine(dicLPSE[100214]);

        // Modification d'un elt
        dicLPSE[100214] = new Etudiant("Duck", "Donald",
        20, 100214);
        // test de cle avec try/catch
        uint codeTest = 100214;
        try
        {dicLPSE.Add(codeTest, new Etudiant("Duck",
        "Donald", 18, 100214));}
        catch (System.ArgumentException)
        { Console.WriteLine("code apogee "+codeTest+ "
        deja utilise"); }
        // alternative possible avec recherche de clé
        Etudiant unEtudiant;
        if (!dicLPSE.TryGetValue(100214, out unEtudiant))
        Console.WriteLine("No val"); // "No val"
        else
        Console.WriteLine(unEtudiant);
        // Methode 1:parcours du dictionnaire pour chaque
        couple cle/valeur
        Console.WriteLine("-----");
        foreach (KeyValuePair<UInt32, Etudiant> kv in
        dicLPSE)
        Console.WriteLine(kv.Key + "; " + kv.Value);
        // Methode 2:parcours du dictionnaire pour chaque
        cle
        Console.WriteLine("-----");
        foreach (UInt32 code in dicLPSE.Keys)
        Console.WriteLine(code+"/"+dicLPSE[code]);
        // Methode 2:parcours du dictionnaire pour chaque
        "valeur"
        Console.WriteLine("-----");
        foreach (Etudiant etud in dicLPSE.Values)
        Console.WriteLine(etud);

        Console.ReadKey();
    }
}
```

Manipulation des données: LINQ

A partir de la version 3.0 de C#, des fonctionnalités supplémentaires sur la manipulation des données ont été rajoutés au langage, regroupées dans l'espace de noms: *Language-Integrated Query*.(LINQ).

L'objectif n'est pas de présenter les fondements de LINQ mais de montrer , par le biais d'exemple, la puissance de ces fonctions associés à des conteneurs. Pour plus de compléments veuillez vous reporter à la bibliographie

Exemple : Dictionnaire et Extraction de données

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Dictionnaire_Compare
{
    public class Etudiant
    {
        public string Nom {get;set;}
        public string Prenom {get;set;}
        public UInt16 Age {get;set;}
        public UInt32 CodeApogee {get;set;}
        public Etudiant(string nom,string prenom,UInt16age, UInt32 codeApogee)
        {Nom=nom; Prenom=prenom;Age=age;CodeApogee=codeApogee;}

        public override string ToString()
        {return (CodeApogee+" "+Nom+" "+Prenom+" age de "+Age.ToString()+" ans");}

    }
    class Program
    {
        static void Main(string[] args)
        { // creation dictionnaire
            Dictionary<UInt32,Etudiant> dicLPSE=new Dictionary<UInt32,Etudiant> ();
            // ajout des entrees
            dicLPSE.Add(100213,new Etudiant("Oscar","Amel",20,100213));
            dicLPSE.Add(110214,new Etudiant("Copter","Elie",21,110214));
            dicLPSE.Add(110216, new Etudiant("aaaa", "bbb", 20, 110216));
            dicLPSE.Add(100215, new Etudiant("aaaa", "dddd", 21, 100215));

            Console.WriteLine("Dictionnaire non trié");
            IEnumerable<KeyValuePair<UInt32, Etudiant>> request1 =
                from keyValue in dicLPSE// pour elt du dictionnaire(donc un couple <cle,valeur> )
                where (keyValue.Value.CodeApogee > 100000) && (keyValue.Value.CodeApogee <110000) // filtre
                select keyValue; // renvoie les <cle,valeur> correspondant
            foreach (var kv in request1) // request1 est une collection de <cle,valeur>
            Console.WriteLine(kv.Value); //kv.Value est une etudiant , appel à la surcharge ToString
            Console.WriteLine("-----");
            var request2 = from keyValue in dicLPSE// var de type IEnumerable<KeyValuePair<UInt32,Etudiant>>
                where (keyValue.Value.Age == 20) // filtre age=20
                orderby keyValue.Value.Nom// trie ascendant du resultat de filtrage
                select keyValue;// where(keyValue.Value.Nom).StartsWith("aa"); // renvoie des valeur keyValue
            foreach (var kv in request2)
            Console.WriteLine(kv.Value);
            Console.WriteLine("-----");
            var request3 = from keyValue in dicLPSE
                // tries enchainés: par age decendant , puis pas nom, puis par prenom
                orderby keyValue.Value.Age descending, keyValue.Value.Nom, keyValue.Value.Prenom
                select keyValue;
            foreach (var kv in request3)
            Console.WriteLine(kv.Value);
            Console.WriteLine("-----");
            var request4 = from keyValue in dicLPSE
                where (keyValue.Value.Nom).StartsWith("aa") // filtre Nom commençant par "aa"
                where (keyValue.Value.Age == 20) //puis filtre Age=20
                select keyValue; // projection des valeurs
            foreach (var kv in request4)
            Console.WriteLine(kv.Value);
            Console.WriteLine("-----");
            var request5 = from keyValue in dicLPSE
                // constitution de groupe en fonction des 2 premier chiffre du code Apogee
                group keyValue by (keyValue.Value.CodeApogee.ToString()).Substring(0, 2) into grps
                select grps; // projection des ssgroupe
            foreach (var ssgroupe in request5) // pour chaque ssgroupe de la requete
            {
                Console.WriteLine(" groupe ayant pour cle " + ssgroupe.Key);
                foreach (var kv in ssgroupe) // pour chaque elt des sousgroupe
                Console.WriteLine(kv.Value); // affiche etudiant
            }
            Console.WriteLine("-----");
            var request6 = from ssgrpe in request5 // pour chaque ssgrpe de la requete 5
                from kv in ssgrpe// pour chaque elt des sous-groupes
                where kv.Value.Age == 20 // ne retient que cerux dont Age=20
                select kv; //projection des elts
            foreach (var kv in request6)
            Console.WriteLine(kv.Value);
            Console.ReadKey();}}

```

TRAVAIL PERSONNEL

1- Manipulation de fichiers

- Ecrire dans un fichier 'texte' les 100 premières valeurs de la fonction $y = \sqrt{x}$
Le format sera: valeur de x;valeur de y \n avec un encodage utf16.
- Ecrire le programme de relecture des valeurs avec un affichage à l'écran
- Refaire la même chose avec un fichier 'binaire'
- Reprendre la question 1 et rajouter un champ date/temps au fichier:
Date/temps; valeur de x;valeur de y
La date initiale sera saisie au clavier, puis à chaque échantillon écrit dans le fichier, le temps sera avancé de 1h
Le format de la date sera de type o (insensible à la culture – voir paragraphe correspondant)

2- Ecrire un client/serveur tcp simplifié

- Tester le bon fonctionnement en local (127.0.0.1) de votre client/serveur envoyant la chaîne de caractère "ON\n" à chaque appui sur la barre espace. Vous utiliserez les méthodes BinaryWriter/ BinaryReader sur un flux NetworkStream pour l'envoi et la réception des données. Vous ouvrirez 2 instances de Visual Studio, une avec le serveur que vous lancerez en premier et l'autre avec le client
- L'ordre de commande est maintenant: "ON;xx\n"; ou "OFF;yy\n" avec xx et yy: valeur de 1 à 10. Le serveur interprétera cette commande (revoir les string et la commande Split) et horodatera la réception commande sous la forme "Lundi 3 Novembre à 10:23. Vous utiliserez la fonction DateTime.now.

3- Conteneur de type List<string>

- Créer une liste avec les sept couleurs de l'arc en ciel
- Afficher la liste par ordre alphabétique (Méthode Sort)
- Rajouter la couleur "marron"
- Afficher le nombre d'élément dans la liste (propriété Count)
- Tester si la couleur "rose" est présente (Méthode Contains)
- En vous inspirant de l'exemple du cours et en utilisant la méthode FindAll:
 - Ecrire la méthode statique permettant de trouver toutes les couleurs ayant 5 lettres
 - Ecrire la méthode statique permettant de trouver toutes les couleurs se terminant par "ge"

4- Travail de recherche: la classe WebClient

Cette class permet de faire une requete sur un serveur http et de récupérer via la méthode DownloadFile le résultat:

Voci un exemple à tester chez vous pour afficher utilise le serveur météo de yahoo (ATTENTION: pas de proxy dans cette exemple)

```
using (WebClient clientWeb = new WebClient())
clientWeb.DownloadFile("http://weather.yahooapis.com/forecastrss?w=628879&u=c",
"page.html"); //628879 code pour toulon
System.Diagnostics.Process.Start("page.htm");
```

Faire une petite application graphique qui affiche la cartographie d'un lieu en se basant sur google map static:

Etape 1: comprendre la requete à envoyer au serveur

avec un navigateur taper <http://maps.google.com/maps/api/staticmap?xxxxxxxx>
xxxxx <http://code.google.com/intl/fr/apis/maps/documentation/staticmaps/>

Etape 2: faire une interface graphique la plus simple: 1 bouton Update+ 1 pictureBox pour afficher le fichier png retourné par le serveur google

La mise à jour de l'image se fera à partir du code suivant à adapter en fonction de vos besoins:

```
fs = new FileStream("img.png", FileMode.Open, FileAccess.Read);
pictureBox1.Image = Image.FromStream(fs);
fs.Close(); // important de relacher la ressource
```

TP

Conteneur de type List< DataCapteur >

On se propose de créer une liste contenant la resultat de différents capteurs. Chaque capteur est repéré par un identifiant unique. Créer la classe DataCapteur suivante

La méthode ToString est surchargée afin d'afficher les informations du capteur.



- 1- Créer la classe et une liste contenant différentes valeurs (pas de saisie clavier, utiliser directement la méthode Add). Vérifier avec un affichage de la liste.
- 2- Rajouter les méthodes *statiques*:
 - a. FindIdEven ,utilisée par la méthode FindAll de la classe List<T>, qui testera la parité de l'identifiant (voir exemple du cours).
 - b. SortById, utilisée par la méthode Sort(Comparison<T>) de la classe List<T>, qui permettra de tri des capteurs dans l'ordre de leur identifiant. Vous utiliserez dans SortById la méthode objetx.ToCompare(objetyy) avec l'exemple: <http://msdn.microsoft.com/en-us/library/w56d4y5z.aspx>
- 3- Faire une requete LINQ pour trier par identifiant descendant puis par valeur ascendante
- 4- Faire une requete LINQ pour regrouper les capteurs par Id puis calculer 'à la main', pour chaque capteur, la moyenne des valeurs de ce capteur.

Client POP3 simple

Le projet suivant consiste, à partir d'un client TCP et en étudiant le protocole POP3 (voir synoptique ci-dessous + <http://www.commentcamarche.net/contents/internet/sntp.php3>), d'envoyer les différentes requetes afin de récupérer les mails sur un serveur POP3 et des afficher en mode console.

Les recommandations:

- Vous pouvez utiliser outlook express (msimn.exe) couplé ArgoSoft.
- Les mails envoyés sont stockés au format .eml dans les répertoires aux noms des utilisateurs que vous aurez créé (voir TP précédent). Vous pouvez directement les supprimer pour faire le 'ménage' au besoin.
- *Le client doit rester simple.* Les pièces jointes ,les encodages MIME de type Base64 ne sont pas traités. Il est donc recommandé de choisir le format d'encodage 'texte brut' sur outlook express.
- La lecture du flux TCP se fera au niveau octet.
- La conversion byte-> string peut se faire par exemple sous cette forme:

```
string code=Encoding.ASCII.GetString(uneliste.ToArray());
```

Client	Mail server	Notes
Client connects...	+OK Hello there.	Welcome message
USER joe	+OK Password required.	
PASS password	+OK Logged in.	
LIST	+OK	Lists the ID and file size of each message on the server
	1 1876	
	2 5412	
	3 845	
	.	
RETR 1	+OK 1876 octets	Retrieves the message with the specified ID
	Content of message #1...	
	.	
DELE 1	+OK Deleted.	Deletes a message from the server
QUIT	+OK Bye-bye.	

Le cahier des charges:

- Une seule contrainte: vous devrez utiliser des listes!
- Vous créerez les classes que vous souhaitez:
Par exemple les classes :
Message: reflète le contenu d'un message. A minima: sa taille plus une liste de chaînes string (info du serveur+ sujet+ body etc...)
POP3Simple qui contiendra par exemple une liste des Messages plus des méthodes telles que login (user,password) etc...

HERITAGE ET DERIVATION

Il est fortement recommandé, pour une bonne assimilation du cours, de reprendre les exemples (copier/coller) et de les tester par vous-même (et si possible en pas à pas).

concept

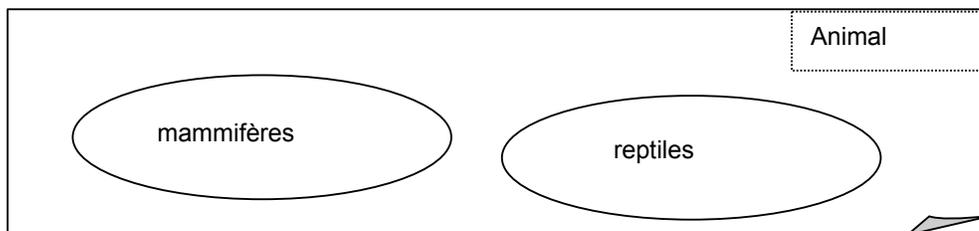
L'héritage permet de construire de nouvelles classes (classe fille) dérivant d'une classe existante (classe mère). La classe dérivée (fille) hérite automatiquement des données et du savoir-faire de la classe mère. Elle peut bien sûr ajouter de nouvelles données et de nouvelles fonctionnalités qui lui sont propres. Ce mécanisme est appelé spécialisation. Un ensemble de classes obtenues par dérivation successive forme une hiérarchie de classe.

La notion d'héritage et de dérivation est à rapprocher de la notion d'ensemble. Cette vision ensembliste permet de définir des sùr-ensembles et des sous-ensembles. La classe fille est un sous-ensemble de la classe mère.

Quelques exemples pour fixer les idées :

Classe de base
Individu
Véhicule
Animal
Animal

En prenant l'exemple de la classe animal graphiquement nous pourrions représenter sous forme d'ensemble :



Que peut-on en conclure :

un mammifère est un animal
un animal n'est pas forcément un mammifère
un reptile est un animal
les animaux possèdent des caractéristiques communes (poids, âge) et des fonctions communes (manger, crier etc...)

Les mammifères, en plus des caractéristiques communes à tous les animaux, possèdent leurs propres caractéristiques ainsi que des méthodes propres.

Héritage ou inclusion :

L'héritage exprime donc une relation du type « est un ».

Par exemple : un chien « est un » mammifère. Une voiture « est un » véhicule.

L'inclusion, c'est-à-dire lorsqu'une classe réutilise pour sa définition une autre classe (exemple d'une classe rectangle utilisant une classe Point), exprime une *relation « possède un » ou « à un »*.

Par exemple : Un rectangle « possède des » points. Une voiture « à un » châssis.

Mis en œuvre

En C# une classe ne peut hériter que d'une autre classe. L'héritage multiple est interdit.

La syntaxe pour exprimer une dérivation est la suivante : class B : A

où A est appelée classe de base, "superclasse", classe mère...

où B est appelée classe dérivée, classe spécialisée, classe fille....

Un exemple pour fixer les idées :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace heritage1
{
    enum RACE { INCONNU, GOLDEN, CAIRN, DANDIE,
    SHETLAND, DOBERMAN, LAB };

    public class Mammifere
    {
        // propriétés automatique
        protected int Age {get;set;}
        protected int Poids {get;set;}
        private int NbrePattes {get;set;}

        // constructeurs
        public Mammifere()
        { Console.WriteLine("Ctor Mammifere 1"); }
        public Mammifere(int f_age, int f_poids, int
        f_nbre_patte)
        {
            Console.WriteLine("Ctor Mammifere 2");
            Age=f_age;Poids=f_poids;NbrePattes=f_nbre_patte;
        }
        public void Crier() {Console.WriteLine("Raaaa");}
        public void Dormir() {Console.WriteLine("Zzzzz");}
        public override string ToString()
        { // NbrePattes inaccessible
        // j'appelle base.ToString() et je rajoute lesinfos
        spécifiques du Chien
        string retourToStringdeLaClasseMammifere =
        base.ToString();
        return ("age "+Age+", poids "+Poids+", pattes
        "+NbrePattes);
        }
    }

    class Chien : Mammifere
    {
        public RACE Race { get; set; }

        public Chien(RACE f_age, int f_age, int
        f_poids,int f_nbre_patte)
        {
            Race = f_age;
            Age=f_age;
            Poids=f_poids;
            //this.NbrePattes=f_nbre_patteompil KO car
            NbrePattes est private!
        }

        public Chien(int f_age, int f_poids,int
        f_nbre_patte):base(f_age,f_poids,f_nbre_patte)
        {
            Race =RACE.INCONNU;
        }

        public override string ToString()
        { // NbrePattes inaccessible
        // j'appellebase.ToString() et je rajoute lesinfos
        spécifiques du Chien
        string retourToStringdeLaClasseMammifere =
        base.ToString();
        return ("Chien " +
        retourToStringdeLaClasseMammifere+ "Race "+Race);
        }
    };

    class Program
    {
        static void Main(string[] args)
        {
            Mammifere unMamifere = new Mammifere(10, 5, 2);
            unMamifere.Crier();
            Console.WriteLine(unMamifere);
            Chien unChien = new Chien(RACE.GOLDEN, 5, 10, 4);
            Console.WriteLine(unChien);
            unChien.Dormir();
            unChien.Crier();
            Chien autreChien = new Chien(2, 1, 4);
            Console.WriteLine(autreChien);

            Console.ReadKey();
        }
    }
}
```

Commentaire :

la classe chien dérive de la classe mammifère. Ceci implique que les fonctions membres de la classe chien peuvent accéder aux champs et propriétés du chien en fonction du niveau de visibilité:

- ✓ public : tout le monde
- ✓ protected : RESTRICTIONS AUX CLASSES DERIVEES
 - protected int Age
- ✓ private: RESTRICTION A LA CLASSE DE BASE
 - private int NbrePattes

```
Ctor Mammifere 2
Raaaa
age 10, poids 5, pattes 2
Ctor Mammifere 1
Chien age 5, poids 10, pattes 0Race
GOLDEN
Zzzzz
Raaaa
Ctor Mammifere 2
Chien age 2, poids 1, pattes 4Race
```

L'idée de base de la dérivation est de pouvoir partager certaines propriétés et certaines méthodes de la classe de base vers les classes dérivées. La classe de base est le "dénominateur" commun d'un ensemble de classe dérivée:

Exemple: unChien.Dormir(); et unChien.Crier

Cependant nous remarquons ici que l'action unChien.Crier ne produit pas une action atisfaisante. La méthode Crier doit être "spécialisée" et adaptée pour refléter le cri d'un Chien: "Ouarffff"

La création d'un objet dérivée, ici un chien, implique une séquence d'initialisation logique:

1. Appel du ctor de chien
2. Appel du ctor de mammifère :*implicite => appel de Mammifere() ou explicite => Chien(yyyyy):base(xxxxxx)*
3. Exécution du code du ctor de mammifère
4. Exécution du code du ctor de mammifère

La spécialisation

Nous avons vu que la classe de base peut intégrer des méthodes communes à toute les classe dérivées. Par exemple la méthode crier de la classe Mammifere peut être vu comme une fonction générique commune à toutes les espèces.

Cependant le cri d'un chat ou d'un chien n'est pas le même évidemment. Il faut donc substituer la fonction crier de base fournie par la classe mère, avec la fonction crier spécifique des classes filles Chat et Chien par exemple.

Ce mécanisme est appelé aussi substitutions de fonction.

Un exemple :

```
namespace heritage1
{
    enum RACE { INCONNU, GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

    public class Mammifere
    {
        // propriétés automatique
        protected int Age { get; set; }
        // constructeurs
        public Mammifere() { Console.WriteLine("Ctor Mammifere 1"); }
        public Mammifere(int f_age)
        {
            Console.WriteLine("Ctor Mammifere 2");
            Age=f_age;
        }
        //methodes
        public void Crier()
        { Console.WriteLine("Raaaa"); }
    }

    class Chien : Mammifere
    { // propriété auto
        public RACE Race { get; set; }
        // ctor 1
        public Chien(int f_age, RACE f_race):base(f_age)
        { Console.WriteLine("Ctor Chien 2"); Race = f_race; ; }

        // specialisation
        // new est optionnel => evite warning
        // compilateur
        // new précise l'intention du concepteur de
        // spécialisé la méthode
        public new void Crier() {
            Console.WriteLine("Ouarffff"); }
    };
}

class specialise
{
    static void Main(string[] args)
    {
        Mammifere unMamifere = new Mammifere(10);
        unMamifere.Crier();
        Chien unChien = new Chien(5, RACE.GOLDEN);
        unChien.Crier();
        Console.ReadKey();
    }
}
}
```

Ctor Mammifere 2
Raaaa
Ctor Mammifere 2
Ctor Chien 2
Ouarfff

Conversion de type

Conversion implicite et explicite

Le langage prévoit la possibilité de référencer :

- ✓ un objet de type classe dérivée à partir d'une référence de Type classe de Base

```
Mammifere unMammifere2 = new Mammifere(5);
Chien unChien2 =new Chien(5, RACE.GOLDEN);
Mammifere autreMammifere2= unChien2; //Upcast
```

Cette conversion de type est appelée "UpCast". Cette conversion réussie toujours à condition de respecter la filiation des classes (conversion dérivée vers Base ou encore classe fille vers classe mère)

- ✓ un objet de type classe de Base à partir d'une référence de Type classe dérivée

```
Mammifere unMammifere2 = new Mammifere(5);
Chien unChien2 =new Chien(5, RACE.GOLDEN);
Chien autreChien2 = (Chien)unMammifere2; // Downcast
```

Cette conversion de type est appelée "DownCasting". Cette conversion NE réussie PAS toujours. Dans ce cas une exception est levée à l'exécution (runtime) car un mammifère n'est pas forcément un Chien, l'inverse par contre étant toujours vrai!
 Cette conversion n'est donc pas de type "safe-fail".

Opérateur is et as

Ces opérateurs sont utiliser pour des conversions de type safe-fail, c'est-à-dire sans levée d'exception au runtime.

L'opérateur as permet de réaliser une conversion de downcasting. La référence résultat est positionnée à null en cas d'échec.

L'opérateur is permet de réaliser un test de conversion. Le resultat, un boolean, est utilisé pour réaliser effectivement la conversion en cas de succès.

```
// downcast avec as
Mammifere unMammifere = new Mammifere(5);
Chien autreChien = unMammifere as Chien;
if (autreChien != null)
    autreChien.Crier();
else
    Console.WriteLine("conversion dwoncast impossible");

// upcast avec is
Chien unChien =new Chien(5, RACE.GOLDEN);
if (unChien is RACE)
    ((Mammifere)unChien).Crier();
else
    Console.WriteLine("conversion upcast impossible");
```

Méthode virtuelle

Une fonction ou une méthode est qualifiée de virtuelle lorsque le mot-clé virtual est en préfixe de sa déclaration.

L'objectif d'une telle déclaration virtuelle est de pouvoir appeler la méthode de substitution adéquate à partir d'une référence de base *pointant vers des objets de type classe dérivée*: ce mécanisme est apeelé polymorphisme (voir paragraphe suivant)

Pour illuster le propos, reprenons l'exemple des mammifères avec 2 classes dérivées: Chat et Chien

```
using System;
using System.Text;

namespace virtuelleexemple
{

    public class Mammifere
    {
        //methodes
        public virtual void Crier() { Console.WriteLine("Raaaa"); }
        public virtual void Sous_Ordre_Especes() { Console.WriteLine("Mammifere"); }
        public void NbreDePattes() { Console.WriteLine("Inconnu"); }
    }

    class Chien : Mammifere
    {
        public override void Crier() { Console.WriteLine("Ouarfff"); }
        public void Sous_Ordre_Especes() { Console.WriteLine("Canidés"); }
    }
}
```

```

    public void NbreDePattes() { Console.WriteLine("Chien: 4 pattes"); }
};

class Chat : Mammifere
{
    public override void Crier() { Console.WriteLine("Miaou"); }
    public override void Sous_Ordre_Especes() { Console.WriteLine("Félinés"); }
    public void NbreDePattes() { Console.WriteLine("Chat: 4 pattes"); }
};

class convert
{
    static void Main(string[] args)
    {Chien unChat = new Chat(); KO impossible
    Mammifere mam1 = new Chat(); // Upcast OK
    mam1.Crier();
    mam1.Sous_Ordre_Especes();
    mam1.NbreDePattes();

    mam1 = new Chien(); // mam1 pointe vers un chien maintenant Upcast OK
    mam1.Crier();
    mam1.Sous_Ordre_Especes();
    mam1.NbreDePattes();
        Console.ReadKey();}
}
}
}

```

Miaou - résultat de mam1.Crier(); mam1 bien qu'étant de type mammifère "comprend" grâce au couple virtual/override que l'objet pointée est un Chat
 Félinés - résultat de mam1.Sous_Ordre_Especes(); même remarque
 Inconnu - résultat de mam1.NbreDePattes(); c'est la méthode de la classe mammifère qui a été appelé!

Ouarfff - résultat de mam1.Crier(); effet de virtual/override en place
 Mammifere - résultat mam1.Sous_Ordre_Especes(); override absent dans la classe dérivée
 Inconnu - résultat de mam1.NbreDePattes(); c'est la méthode de la classe mammifère qui a été appelé!
virtual ABSENT sur NbreDePattes (classe Mammifere)
virtual ABSENT sur NbreDePattes (classe Mammifere)

Polymorphisme

Plusieurs objets de natures différentes peuvent avoir des fonctionnalités similaires mais implémentées différemment au sein de chaque objet (c'est la spécialisation).

Le cas précédent en est un exemple : tous les mammifères ont un cri mais qui diffèrent finalement en fonction de chaque espèce.

Le polymorphisme (« qui possède plusieurs formes »), permet de donner un même nom à chacune de ses actions. La bonne action sera sélectionnée contextuellement en fonction de la classe d'objets à laquelle elle s'applique.

L'intérêt de cette approche est de pouvoir unifier un traitement appliqué à des objets de nature différente: En reprenant l'exemple précédent des mammifères:

```

// On regroupe tout le monde sous le type de base
List<Mammifere> maMenagerie=new List<Mammifere>();
// Je peuple la ménagerie
maMenagerie.Add(new Chat());
maMenagerie.Add(new Chien());
maMenagerie.Add(new Mammifere());
// Je fais crier tous le monde
foreach (Mammifere mam in maMenagerie)
    mam.Crier();

```

Miaou
 Ouarfff
 Raaa

TRAITEMENT DE BASE APPLIQUE A DES OBJETS DIFFERENTS(MAIS APPARTENANT A LA MÊME CLASSE DE BASE)

Exemple d'application polymorphique

Prenons l'exemple d'une application manipulant un ensemble d'objets graphiques. Chacun de ces objets possède une méthode de calcul de surface et de périmètre.

La généralisation du concept d'objets graphiques aboutit à une classe *ObjetGraphique* contenant les fonctions virtuelles surfaces et périmètres.

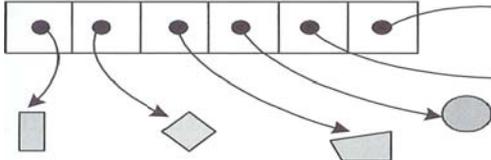
Il est possible alors de stocker tous les objets manipulés par l'application dans un tableau du type *ObjetGraphique*.

```

ObjetGraphique [] monTableau =.....
monTableau[0]=new Rectangle(....)
monTableau[1]=new Cercle(....)
monTableau[2]=new Cercle(....)
foreach (ObjetGraphique obj in monTableau)
    obj.CalculSurface();

```

Chaque case du tableau est une référence sur des objets différents mais que appartiennent à la même famille.



Grâce à la virtualisation les méthodes de surface et périmètre sont polymorphiques (codes différents en fonction de la nature de l'objet): la fonction adéquate est automatiquement appelée.

Le polymorphisme fournit un niveau d'abstraction élevée : il est possible de demander un objet graphique de calculer son périmètre où sa surface sans avoir besoin de connaître précisément sa classe réelle.

Les avantages :

- possibilité de définir un corps différent pour une fonction pour des classes différentes
- manipulation d'un ensemble d'objets de façon générique sans connaître leur type
- possibilité d'ajouter des classes dérivées supplémentaires sans retoucher au code.

La classe object

Il s'agit de la classe de base fondamentale parmi toutes les classes du .NET Framework. La plupart des classes que vous manipulez héritent de « object ». Cette classe constitue la racine de la hiérarchie des types.

```

public class Object
{
    public Object();
    public extern Type GetType();
    public virtual bool Equals (object obj);
    public static bool Equals (object objA, object objB);
    public static bool ReferenceEquals (object objA, object objB);
    public virtual int GetHashCode();
    public virtual string ToString();
    protected override void Finalize();
    protected extern object MemberwiseClone();
}

```

Comme toutes les classes du .NET Framework sont dérivées de **Object**, toutes les méthodes définies dans la classe **Object** sont disponibles dans la totalité des objets du système. Les classes dérivées peuvent substituer certaines de ces méthodes, notamment :

- **Equals** - Prend en charge les comparaisons entre objets.
- **Finalize** - Effectue des opérations de nettoyage avant qu'un objet soit automatiquement récupéré.
- **GetHashCode** - Génère un nombre correspondant à la valeur de l'objet pour prendre en charge l'utilisation d'une table de hachage.
- **ToString** - Fabrique une chaîne de texte explicite qui décrit une instance de la classe.

Compléments :

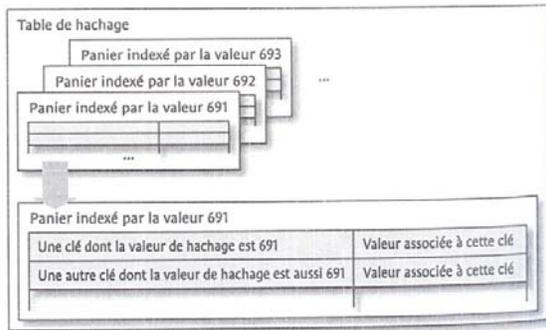
ToString : Par défaut la méthode ToString appliquée à une de vos propres classes renverra le type sous forme de chaîne de caractère. Vous comprenez maintenant mieux la syntaxe utilisée depuis le début pour personnaliser l'affichage de vos classes :

```

public override string ToString() // spécialisation de la classe
{
    // base.ToString() ici appelle ToString de la classe mère
    // +" Auteur: "+Auteur; et on ajoute les infos spécifiques à la classe fille
    string chaine=base.ToString()+" Auteur: "+Auteur;
    return (chaine);
}

```

GetHashCode : certains conteneurs (Dictionary, SortedList ..) utilise une clé dite de hachage, en fait tout simplement un entier, pour indexer les éléments du conteneur. Tous les éléments ayant une même clé sont regroupés dans un même panier (bucket). La clé de hachage, qui n'est pas unique mais suffisant « distincte » ne doit pas être confondu avec la clé d'identification d'un dictionnaire, qui elle est unique et qui permet de retrouver l'élément dans un panier.



Hash code : identifiant « panier »
 Possible d'avoir 2 paniers indexés 691 !!!

Couple <Key,Data> :
 Key : clé **unique** d'identification de la valeur
 Data : valeur que l'on souhaite stocké

Comment construire le « hash code ».

Afin d'optimiser les algorithmes de recherches et d'indexation, votre méthode **GetHashCode** doit renvoyer un entier « suffisamment unique ». Il n'y pas de règles absolues mais l'examen de vos classes permet de vous donner des pistes.

Exemples divers pour vous aider :

```
public override int GetHashCode()
{
    return _unChampInt.GetHashCode();
}
-----
public override int GetHashCode (Customer obj)
{
    return (obj.prenom + ";" + obj.nom).GetHashCode();
}
-----
public override int GetHashCode() {
    return coordx ^ coordy; // OU exclusif
}
-----
```

```
public override int GetHashCode () {
    return a.GetHashCode() ^ b.GetHashCode() ^
    c.GetHashCode();
}
-----
public override int GetHashCode()
{
    return Measure2 * 31 + Measure1; // 31 = un nombre
    premier
}
-----
```

Classe abstraite

Une classe est dite abstraite lorsque le préfixe `abstract` est rajouté devant le mot clé `class` :

```
abstract class A_Fiche
{
    Protected int valeur=10;
    public void test() { Console.WriteLine("test"); }
    public int AccessValeur { get; set; }
}
```

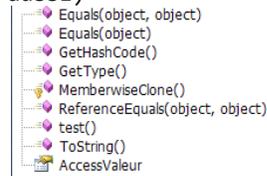
Une classe abstraite ne peut être instantiée directement :

```
var test = new A_Fiche1(); // KO
```

SOLUTION

```
class Fiche1 : A_Fiche
{
}
```

`var ref_Fiche = new Fiche1();`
`ref_Fiche` hérite de `A_Fiche` (et de `Object` aussi)



Une classe abstraite permet de déclarer des membres abstraits (sans corps de fonction).

Les classes dérivées héritant de la classe abstraite SONT OBLIGES d'implémenter le corps de la méthode.

```
abstract class A_Fiche
{
    protected int valeur=10;
    public void test()
    { Console.WriteLine("test"); }
    abstract public int AccessValeur
    { get; set; } // PAS DE CORPS!!!!
}
class Fiche1 : A_Fiche1
{
    //OBLIGATION d'IMPLEMENTER l'ACCESSEUR!
    public override int AccessValeur
    {
        get { return (valeur); }
        set { valeur = value; }
    }
}
```

REMARQUE : le fait de déclarer une méthode abstraite ouvre droit au polymorphisme (équivalent à écrire `virtual`). Toute les fiche peuvent être manipulée comme des `A_Fiche`

Limite de l'héritage

Introduction

Les notions de dérivation et d'héritage permettent de construire plus aisément de nouvelles classes en s'inspirant d'un modèle de base (la classe que l'on dérive). Ainsi, plutôt que de partir de rien, la nouvelle classe dérivée hérite d'un certain nombre de fonctionnalités. L'objectif est donc de structurer le programme et de factoriser le code en exploitant la relation "EST un" : un chien EST UN mammifère.

Cependant l'héritage n'est pas la panacée et certaines limites existent.

Critique de l'exemple précédent

Comment traiter alors une Baleine qui est un mammifère aussi mais qui n'a pas de patte?

Nous souhaitons traiter la capacité des mammifères à voler? Faut t'il mettre voler dans la classe de base?

Voici un exemple de modifications apportés à l'exemple:

```

public class Mammifere
{
//methodes
public virtual void Crier() { Console.WriteLine("Raaa"); }
public virtual void Sous_Ordre_Especes() { Console.WriteLine("Mammifere"); }
public virtual void NbreDePattes() { Console.WriteLine("Inconnu"); }
}

class Baleine : Mammifere
{
    public override void NbreDePattes() { Console.WriteLine("je n'ai pas de patte");}
    public override void Crier() { Console.WriteLine("Mouuuuu"); }
    public override void Sous_Ordre_Especes() { Console.WriteLine("Cétacés");}
}

class ChauveSouris : Mammifere
{
    public override void NbreDePattes() { Console.WriteLine("je n'ai pas de patte");}
    public override void Crier() { Console.WriteLine("Mriiiii"); }
    public override void Sous_Ordre_Especes() { Console.WriteLine("chiroptères");}
    public void Vole() { Console.WriteLine("Flap flap");}
}

class convert
{
    static void Main(string[] args)
    {
        ChauveSouris uneChauveSouris=new ChauveSouris();
        uneChauveSouris.Vole(); //OK car uneChauveSouris du type uneChauveSouris
        // On regroupe tout le monde sous le type de base
        List<Mammifere> maMenagerie=new List<Mammifere>();
        // Je peuple la ménagerie
        maMenagerie.Add(new ChauveSouris());
        // maMenagerie[0].Vole(); KO maMenagerie[0] fait référence à un mammifere
        // Vole() n'est pas dans la classe de base!
        maMenagerie.Add(new Baleine());

        //Qui vole dans ma ménagerie???
        foreach (Mammifere mam in maMenagerie)
            // mam.Volant(); KO: volant n'est pas dans la classe de base!
    }
}

```

Remarques: Nous constatons que la spécialisation rajoutée à la chauve-souris ne permet pas d'appliquer un traitement de base du type: qui vole? (polymorphisme). Une solution consiste à intégrer Vole() dans mammifère en virtual. Ceci nous amènera à surcharger toutes les classes dérivées. Le code produit n'est donc pas robuste et une modification du cahier des charges implique une retouche de toutes les classes.

La question est alors de savoir comment injecter des comportements à des objets non liés spécifiquement à la classe de base (un animal volant n'est pas une spécificité des mammifères).

Les "interfaces" en C# répondent en autre à cette question. Les interfaces définissent un certain de nombre de fonctionnalité que les classes choisissent ou non d'implanter. Cette notion d'interface sera détaillée plus loin dans ce chapitre

Voici l'exemple réécrit avec des interfaces:

```

//Fonction Vole
interface IVolant
{void Vole();}

// Fonction a des pattes
interface IPatte
{ // définition d'une propriété en lecture
  int NbPatte { get; }
}

// classe de base
public class Mammifere
{
  //methodes communes à tous les mammifères
  public virtual void Crier() { Console.WriteLine("Raaaa"); }
  public virtual void Sous_Ordre_Especes() { Console.WriteLine("Mammifere"); }
}
// Chien est un mammifère qui a aussi des pattes
class Chien : Mammifere,IPatte
{ // Implantation de NbPatte défini dans IPatte
  public int NbPatte
  {
    get {return(2);}
  }
  // surcharge polymorphe des classes de bases
  public override void Crier() { Console.WriteLine("Ouarfff"); }
  public override void Sous_Ordre_Especes() { Console.WriteLine("Canidés"); }
}

class Baleine : Mammifere
{ // surcharge polymorphe des classes de bases
  public override void Crier() { Console.WriteLine("Mouuuuu"); }
  public override void Sous_Ordre_Especes() { Console.WriteLine("Cétacés");}
}

// Chien est un mammifère avec les fonctionnalités: pattes + volant
class ChauveSouris : Mammifere,IVolant,IPatte
{ // Implantation de NbPatte défini dans IPatte
  public int NbPatte
  {
    get {return(2);}
  }
  // Implantation de Vole défini dans IVolant
  public void Vole() { Console.WriteLine("Flap flap"); }
  // surcharge polymorphe des classes de bases
  public override void Crier() { Console.WriteLine("Mriiiiii"); }
  public override void Sous_Ordre_Especes() { Console.WriteLine("chiroptères");}
}

}

class convert
{
  static void Main(string[] args)
  {
    ChauveSouris uneChauveSouris = new ChauveSouris();
    uneChauveSouris.Vole();
    // On regroupe tout le monde sous le type de base
    List<Mammifere> maMenagerie = new List<Mammifere>();
    // Je peuple la ménagerie
    maMenagerie.Add(new ChauveSouris());
    maMenagerie.Add(new Baleine());
    maMenagerie.Add(new ChauveSouris());
    maMenagerie.Add(new Chien());
    //Qui vole dans ma ménagerie???)
    foreach (Mammifere mam in maMenagerie)
    {
      if (mam is IVolant)
        ((IVolant)mam).Vole();
      else
        Console.WriteLine("Non Volant");
    }
    //Allez tous le monde crie
    foreach (Mammifere mam in maMenagerie)
      mam.Crier(); // triatement de base crier =>polymorphisme
  }
}

```

Alternative à l'héritage

Cette partie n'est pas abordée dans notre cours car elle implique une bonne maîtrise des notions d'interfaces, d'héritage. Des indications sont données pour ceux qui devront dans leur profession approfondir les connaissances présentés ici et mettre en œuvre des techniques professionnelles de programmation.

Voici quelques extraits et liens vers des informations sur ces sujets

PatrickSmacchia (article "assez pointu")

<http://www.dotnetguru.org/articles/dossiers/heritageimpl/FragileBaseClass.htm>

"Les deux techniques alternatives à l'héritage d'implémentation pour réutiliser du code dans un programme écrit dans un langage orienté objet sont :

- La composition d'objets : un objet (l'objet encapsulant ou outer object en anglais) en encapsule un autre (l'objet encapsulé ou inner object en anglais). L'objet encapsulant ne doit jamais communiquer à l'extérieur une référence vers l'objet encapsulé. Une conséquence est que l'objet encapsulant est responsable du cycle de vie de l'objet encapsulé. Une autre conséquence est que différents objets encapsulant, éventuellement instances de classes différentes, peuvent utiliser une même implémentation pour leurs objets encapsulés. A l'inverse, une même implémentation d'objets encapsulant peut utiliser différentes implémentations d'objets encapsulés. Il y a donc bien ici possibilité de réutiliser du code, soit du côté de l'implémentation de l'objet encapsulé, soit du côté de l'implémentation de l'objet encapsulant.
- Les interfaces : Lorsque les interfaces sont utilisées conjointement avec la composition d'objets, on peut alors réutiliser du code tout en bénéficiant du polymorphisme que l'on avait avec l'héritage d'implémentation (i.e, on ne connaît pas exactement l'implémentation que l'on utilise, on sait juste quelle respecte un certain contrat, matérialisé par l'interface en l'occurrence). En outre si une interface est suffisamment explicite, il n'est pas nécessaire d'aller analyser le code de ses implémentations pour l'utiliser correctement. Ce phénomène est connu sous le nom de black-box reuse, en opposition au terme white-box reuse déjà cité.

Site du zero :Les limites de l'héritage : le pattern strategy

Article très clair montrant l'implantation de fonctionnalité variant dans le temps à l'aide de la composition d'objet couplée aux interfaces

Les designs patterns

Un livre de référence : Design Patterns: Elements of Reusable Object-Oriented Software écrit par ceux surnommés "The **Gang of Four**(Gof)" (utilisez les mots clés Gof et **designs patterns**)
Indispensable pour une programmation "professionnelle".

Les interfaces

Concept

La notion d'interface rejoint en partie la notion de de classe abstraite. Une classe abstraite est une classe marqué **abstract** qui définit un ensemble de fonctionnalités (méthodes) sans corps associé. Une classe concrète est une classe pour laquelle l'ensemble des fonctionnalités jusqu'alors abstraites ont toutes été concrétisées. Une fonctionnalité concrète est une fonction membre qui possède un corps (c'est à dire du code).

Une **interface** peut être vue donc comme un modèle de classe qui contient *les prototypes* des méthodes et des propriétés (attributs qui servent à accéder à d'autres). Chaque classe qui choisira d'implémenter l'interface devra respecter le contrat imposé et fournir une implémentation de l'interface (ie fournir le code des méthodes) .

Exemple

```
interface IVolant
{void Vole();}

// Fonction a des pattes
interface IPatte
{ // définition d'une propriété en
lecture
  int NbPatte { get; }
}

class ChauveSouris : Mammifere,IVolant,IPatte
{ // Implantation de NbPatte défini dans IPatte
  public int NbPatte
  {
    get {return(2);}
  }
  // Implantation de Vole défini dans IVolant
  public void Vole() { Console.WriteLine("Flap flap"); }
  .....
}
```

Remarques:

- par convention le nom donné aux interfaces est préfixé par un I : `IVolant`, `IPatte`, `Ixxxxx`
- les interfaces n'ont pas de constructeurs
- les interfaces ne définissent pas de code des méthodes (concept abstraction)
- les interfaces n'ont pas de champs
- Les méthodes sont toutes publiques par défaut

Intérêt des interfaces

Les interfaces , lors de l'étude d'un projet, permettent de définir en amont les fonctionnalités attendues des objets sans pour autant figer ou imposer de solutions techniques.

Une fois définies, elles obligent toutes les classes clientes de ces interfaces à respecter le contrat : dans un travail en équipe par exemple l'architecte logiciel ayant défini les interfaces , les objets et leurs interactions les programmeurs devront respecter les choix imposés (conventions, méthodes, fonctionnalités)

L'héritage multiple étant interdit (choix des concepteurs du C#), les interfaces permettent de décrire des fonctionnalités, non spécifique à la classe de base ,attendues des objets.

Exemple

- ✓ Tous les mammifères ne sont pas volants: fonction spécifique des chauve-souris (interfaces `IVolant`)
- ✓ Les ouvrages (livres, DVD) d'une bibliothèque ne sont pas tous empruntables (interfaces `IEmpruntable` qui contiendra sans doute un historique des emprunts). Par contre tous les ouvrages ont un genre, possède un identifiant unique, une date de création etc (=> invariant dans le temps=> classe de base)
- ✓ Des formes en 2D ont toutes la faculté de renvoyer un périmètre, une surface (=> classe de base). Certaines formes sont définies à l'aide d'une liste de point (rectangle , carré, losange, triangle), d'autre non (cercle)=> interface `Ipoint`

Le polymorphisme d'interface

Les interfaces permettent au polymorphisme d'opérer de manière très simple en manipulant les objets au travers d'une référence de type interface.

Exemple 1:

```
List<Ivolant2> maMenagerieVolante = new List<
Ivolant2>();
// Je peuple la ménagerie
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Canari());
maMenagerie.Add(new Perroquet());
maMenagerie.Add(new pterodactyle ());
foreach (Ivolant2 animalVolant in
maMenagerieVolante)
    animalVolant.VitesseDePointe();
```

On constate qu'il est possible de regrouper des objets n'appartenant pas à la même classe de base!! (Canari n'est pas un mammifère). Il suffit juste que la classe de tous ces objets implémente Ivolant2.

Remarque : il faut bien comprendre `List<Ivolant2> maMenagerieVolante` maintient une liste de référence vers des objets en mémoire. Ces objets, lors de leur création, ne sont pas tronqué en mémoire.

`maMenagerie.Add(new ChauveSouris());` réalise 2 opérations :

- Création d'un objet complet en mémoire de type Chauve-souris
- Stockage dans liste d'une référence vers cet objet en tant que `Ivolant2`

Tant que cette chauve souris est manipulée en tant que `Ivolant2` seule les propriétés et méthodes des `Ivolant2` seront accessibles.

A tout moment il est possible de changer de point de vue par une opération de cast pour accéder pleinement à l'objet : `(ChauveSouris)maRéférenceIvolant2` à condition bien entendu que `maRéférenceIvolant2` référence à l'instant *t* une chauve-souris !!!

Exemple 2:

```
List<Mammifere> maMenagerie = new List<Mammifere>();
// Je peuple la ménagerie
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Baleine());
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Chien());
//Qui vole dans ma ménagerie????
foreach (Mammifere mam in maMenagerie)
{
    if (mam is IVolant)
        ((IVolant)mam).Vole();
    else
        Console.WriteLine("Non Volant");
}
//Allez tous le monde crie
foreach (Mammifere mam in maMenagerie)
    mam.Crier(); // traitement de base crier =>polymorphisme
```

On constate qu'il est possible de regrouper des objets n'implémentant pas tous l'interface Ivolant, et de retrouver puis d'effectuer un traitement commun à tous les volants (!en utilisant les cast!)

Interface Explicite et Résolution de conflit

Il est possible de résoudre les conflits de noms en explicitant, lors de l'implantation de la méthode, le nom de l'interface.

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }
public class Widget : I1, I2
{
    public void Foo ()
    {Console.WriteLine ("Widget's implementation of I1.Foo");}
    int I2.Foo()
    {
        Console.WriteLine ("Widget's implementation of I2.Foo");
        return 42;
    }
}
```

```
Widget w = new Widget();
w.Foo(); // Widget's implementation of I1.Foo
((I1)w).Foo(); // Widget's implementation of I1.Foo
((I2)w).Foo(); // Widget's implementation of I2.Foo
```

Interface *IEquatable* et *IEqualityComparer*

Le framework .NET fournit un nombre important d'interface répondant à des problèmes classiques de l'algorithmique:

- Comment cloner des objets *ICloneable<T>*
- Comment parcourir une liste : *IEnumerable<T>*
- Comment tester l'égalité des objets: *IEquatable<T>* et *IEqualityComparer<T>*

Comment comparer des objets: *IComparable<T>* et *IComparer<T>*

IEquatable<T>

Le comportement par défaut d'un test d'égalité $x==y$ ou $x.Equals(y)$ est:

- Pour les types valeurs: vrai si les données sont identiques
- Pour les types valeurs: vrai si les références sont identiques! (et non pas le contenu des objets)

L'interface *IEquatable* définit la méthode *Equals*, qui détermine l'égalité des instances du type d'implémentation.

```
public interface IEquatable<T>
{
    bool Equals (T other);
}
```

Exemple:

```
public class Temperature : IEquatable<Temperature>
{
    // Implémentation de IEquatable
    //paramètre <T> de type Temperature
    public bool Equals(Temperature other)
    { // appel de méthode par défaut Equals des types par valeurs double.Equals
      return m_Temperature_Moyenne.Equals(other.m_Temperature_Moyenne);
    }
    // CHAMPS
    protected double m_Temperature_Moyenne = 0.0;
    //ACCESSEURS
    public double Kelvin
    {
        get
        { return m_Temperature_Moyenne; }
        set
        {
            if (value < 0.0)
                throw new ArgumentException("Temperature >0 svp");
            else
                m_Temperature_Moyenne = value;
        }
    }
}
// CTOR
public Temperature(double degreesKelvin1, double degreesKelvin2)
{
    this.Kelvin = (degreesKelvin1 + degreesKelvin2) / 2;
}
}

public class Example
{
    public static void Main()
    {
        List<Temperature> temps =
            new List<Temperature>();

        temps.Add(new Temperature(2017.15,1000));
        temps.Add(new Temperature(0,100));
        temps.Add(new Temperature(50,50));

        foreach (Temperature t in temps)
            if (t.Equals(temps[1]))
                Console.WriteLine("température moyenne égale");
            else
                Console.WriteLine("température moyenne différente");

        Console.ReadKey();
    }
}
```

température moyenne différente
température moyenne égale
température moyenne égale

IEqualityComparer<T>

Cette interface est utilisée par certains d'algorithmes travaillant avec les conteneurs et en particulier les requêtes LINQ. Alors que `IEqualityComparer` peut-être vu comme le comportement par défaut de l'égalité d'un certain type d'objet, *IEqualityComparer* permet de mettre en place plusieurs scénarios différents d'égalité au travers par exemple de requête LINQ.

IEqualityComparer prévoit l'implémentation de la comparaison d'égalité personnalisée pour le type T:

```
bool Equals(T x, T y).
```

Vous noterez au passage la différence avec le prototype `bool Equals (T other)`

```
public interface IEqualityComparer<T>
{
    bool Equals (T x, T y);
    int GetHashCode (T obj);
}
```

Exemple: Soit la méthode d'extensions de la classe Stack

```
public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source,
    IEqualityComparer<TSource> comparer
)
```

Lecture du prototype

- `IEnumerable<TSource>`: qui renvoie une référence vers un type `IEnumerable` (autrement dit une collection "parcourable" avec un `foreach`)
- `Distinct<TSource>`: Méthode `Distinct` ayant <pour paramètre un objet T>
- `this IEnumerable<TSource> source`: `this` indique la syntaxe spécifique aux méthodes d'extensions. La source sur laquelle porte notre requête doit implémenter l'interface `IEnumerable`. C'est le cas ici puisque nous travaillons avec une pile `public class Stack<T>: IEnumerable<T>, ICollection, IEnumerable`
- `IEqualityComparer<TSource> comparer`: référence vers un objet implémentant `IEqualityComparer` donc par la même la méthode `bool Equals (T x, T y)`

▪ ;

```
public class TemperatureComparerClass :
    IEqualityComparer<Temperature>
{ // Scénario différent de IEquatable: Température égale si
  T1.1=T2.1 et T1.2=T2.2
  public bool Equals(Temperature temp1, Temperature temp2)
  {
    if ((temp1.Temperature_Capteur1 ==
temp2.Temperature_Capteur1) &&
        (temp1.Temperature_Capteur2 ==
temp2.Temperature_Capteur2))
        return true;
    else return false;
  }
  public int GetHashCode(Temperature temp)
  { // utilisé par Dictionnaire ou SortedList pour définir une clé
    INTERNEd e rangement
    // pas d'obligation d'être unique (mais suffisamment différente)
    // GetHashCode <> de Key!!! Key doit être unique car clé de
recherche de vos données
    return ((int)(temp.Temperature_Capteur1 *
temp.Temperature_Capteur2));
  }
}
```

```
public class Temperature : IEquatable<Temperature>
{
  // Implémentation de IEquatable
  //paramètre <T> de type Temperature
  public bool Equals(Temperature other)
  { // appel de méthode par défaut Equals des types par valeurs
double.Equals
    return
Temperature_Moyenne.Equals(other.Temperature_Moyenne);
  }
  // propriétés AUTO
  public double Temperature_Moyenne { get; set; }
  public double Temperature_Capteur1 { get; set; }
  public double Temperature_Capteur2 { get; set; }

  // CTOR
  public Temperature(double degreesKelvin1, double
degreesKelvin2)
  {
    Temperature_Capteur1 = degreesKelvin1;
    this.Temperature_Capteur2 = degreesKelvin2;
    Temperature_Moyenne = (degreesKelvin1 + degreesKelvin2) /
2;
  }
}
```

```
public class Example
{
  public static void Main()
  {
    Stack<Temperature> temps =
    new Stack<Temperature>();
    temps.Push(new Temperature(2017,1000));temps.Push(new Temperature(0, 100));temps.Push(new Temperature(2017, 1000));
    temps.Push(new Temperature(50, 75)); temps.Push(new Temperature(100, 0));
    foreach (Temperature t in temps)
    {
      Console.WriteLine("Compare {0} {1} avec ref {2} {3}",
        t.Temperature_Capteur1, t.Temperature_Capteur2,
        temps.Peek().Temperature_Capteur1, temps.Peek().Temperature_Capteur2);
      if (t.Equals(temps.Peek())) // Peek renvoie l'objet au dessus de la pile
        Console.WriteLine("température moyenne égale");
      else
        Console.WriteLine("température moyenne différente");
    }
    // Mise en place d'un scénario différent pour les requetes LINQ
    IEnumerable<Temperature> tempSansDoublons=Enumerable.Distinct(temps, new TemperatureComparerClass());
    var requete2 = tempSansDoublons.Where(temp => ( temp.Temperature_Capteur1 > 50) &&
        (temp.Temperature_Capteur1 > 25)
        ));
    foreach (Temperature t in requete2)
      Console.WriteLine("température capt1:{0} capt2:{1}",t.Temperature_Capteur1,t.Temperature_Capteur2);
    Console.ReadKey(); }
}
```

TRAVAIL PERSONNEL

Ex : Copier le programme ci-dessous. Exécutez le en mode pas à pas ,indiquez pour chaque ligne la valeur prise par le champ valeur et commentez en conséquence ce programme.
Remplacer le mot clé override dans Fiche2 par new. Que se passe t'il ?

```
abstract class A_Fiche
{
    protected int valeur=10;
    public void test() { Console.WriteLine("test"); }
    abstract public int AccessValeur { get; set; }
}

class Fiche1 : A_Fiche
{
    public override int AccessValeur
    {
        get { return (valeur); }
        set { valeur = value; }
    }
}

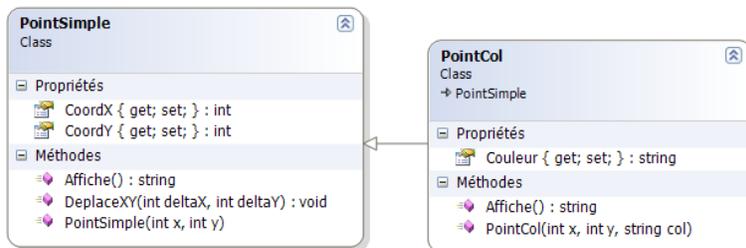
class Fiche2 : Fiche1
{
    public override int AccessValeur
    {
        get { return (valeur); }
        set { valeur = 2*value; }
    }
}

class abstrait1
{
    static void Main(string[] args)
    {
        Fiche1 ref_Fiche = new Fiche1();
        A_Fiche1 ref_A_Fiche1 = ref_Fiche;
        ref_A_Fiche1.AccessValeur = 20;

        Fiche2 ref_Fiche2 = new Fiche2();
        ref_Fiche = ref_Fiche2;
        ref_Fiche.AccessValeur = 2;

        ref_A_Fiche1 = ref_Fiche2;
        ref_A_Fiche1.AccessValeur = 60;
    }
}
```

Ex1 :
créer une classe pointcol, dérivé de PointSimple, comme suit :



Les méthodes de la classe de base ne sont pas virtuelles.
Affiche est spécialisée (surchage) dans la classe dérivée PointCol. Vous utiliserez le mot clé *base* pour faire appel à des méthodes de la classe de base (constructeur de PointCol et Affiche)

Tester vos classes en créant un point ,un point couleur et en appelant les méthodes.

Faire une opération de downcast : PointSimple unPtcast = unPtCol; et tester
Console.WriteLine(unPtcast.Affiche());
Que ce passe t'il ?

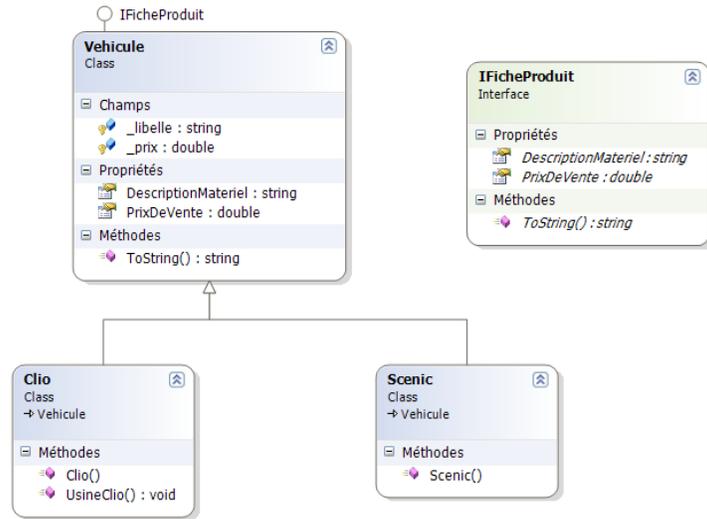
Créer un tableau de PointSimple où seront stockés des PointSimple mais aussi des PointCol. Balayer le tableau et appliquer à tous les objets la méthode Affiche.
 Apporter les modifications nécessaires à votre programme pour que l'affichage garde son sens en fonction de l'objet en cours.
 Comment s'appelle le mécanisme que vous avez mis en œuvre ?

Ex2

Cet exercice sera rcompléter en TP dans le cadre de l'étude du design patern décorateur.

Codez le diagramme suivant :

- Le ctor de Clio et Scenic initialise les champs `_libelle` avec le nom de la voiture (clio ou scenic) et le champ `_prix` avec un prix de vente que vous choisissez.
- La méthode `UsineClio` effectue un simple affichage « Biilancourt-France » par exemple.
- La méthode `ToString` réalise un affichae du prix de vente et du libellé de la voiture.



Créez une clio et testez les méthodes.

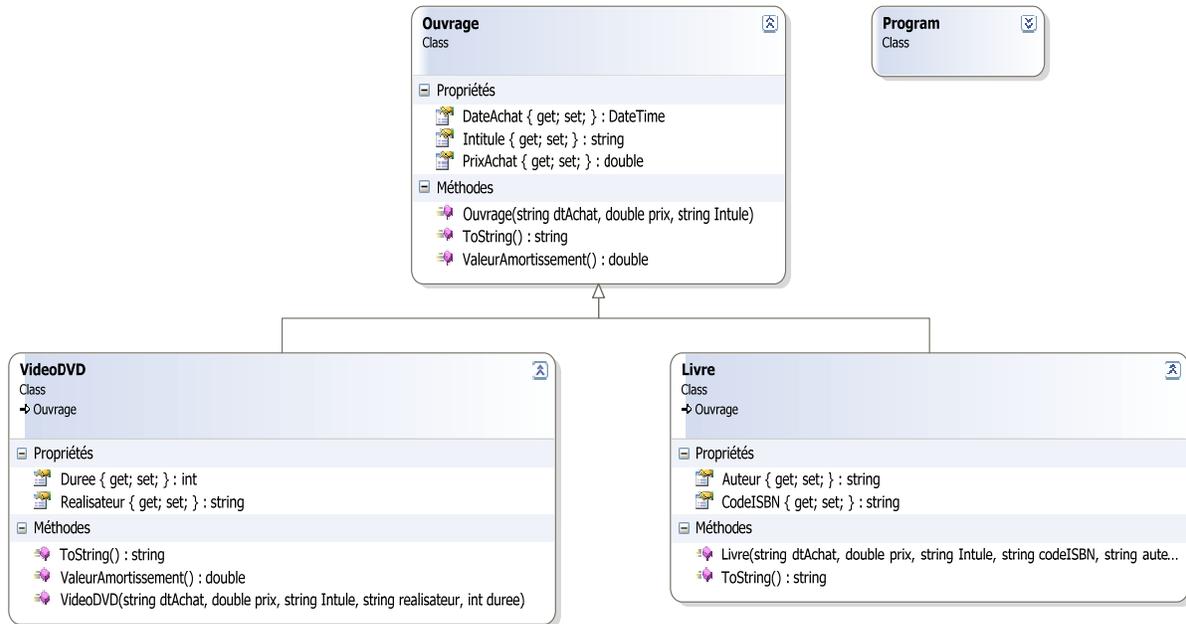
Créez une liste de *IFicheProduit* : ajouter 2 scenic et 1 Clio

- Balayer la liste et afficher tous les voitures de la liste.
- Rajouter en suite l'affichage de l'endroit de fabrication des Clio (utilisez les opérateurs `as` ou `is`).

TP

Ex1 :

Soit la hierrarchie des classes suiavntes



La méthode VIRTUELLE `ValeurAmortissement()` d'`Ouvrage` permet de calculer la valeur d'un ouvrage en fonction de son age selon la formule $(1-x\%)^d$ avec d = jour écoulé entre date actuelle- date d'achat et x : pourcent de dépréciation de l'ouvrage chaque jour (par exemple 0,1%):
Vous pourrez utiliser: `Math.Pow`, `TimeSpan` pour la différence et la propriété `xx.Days` qui renvoie les jours

La méthode `ToString` de la classe de base, surchargée, renvoie une chaîne avec la date d'achat .

Les méthodes `ToString` des classes dérivées complètent l'information avec leurs champs spécifiques en réutilisant `ToString` de `Ouvrage` (`String chaîne = base.ToString() + " Réalisateur: "....`)

Faire un programme qui permette de créer une liste d'ouvrage. Peupler cette liste.

Balayer la liste ainsi créée et :

- Afficher l'ouvrage en cours : `Console.WriteLine(ouvrage);` //appel de `ToString`
- Calculer la valeur de l'ouvrage
- Mettre à jour la valeur globale du stock

Afficher la valeur totale de votre stock.

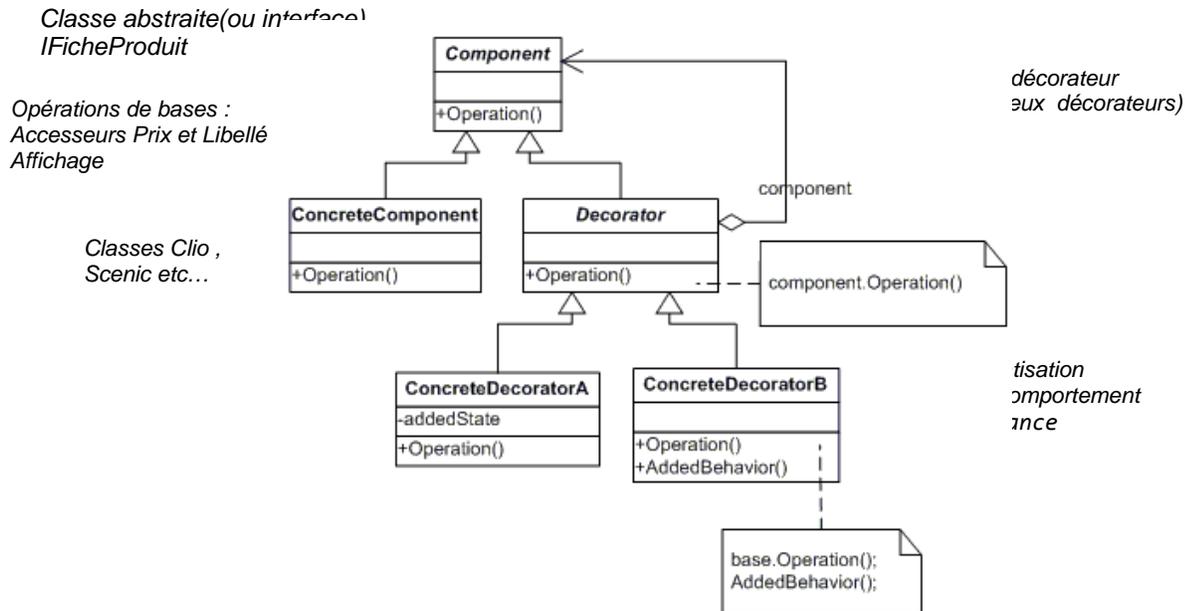
TP : Etude du design pattern décorateur

Veillez lire les documentations fournies en TP sur ce pattern.

L'exercice 2 de la préparation est repris. On souhaite traiter le problème des options rajoutées aux véhicules : par exemple climatisation, peinture métallisée, intérieur en bois etc.

En ne tenant compte que des 3 options citées nous aboutissons à 9 combinaisons possibles ! La solution consistant à dériver de la classe clio toutes les possibilités (clioavecclim dérive clio) n'est pas viable car elle entraîne une explosion combinatoire des classes.

Le design pattern « Décorateur » propose une solution basée sur le schéma UML suivant :



Les points clés :

*MISE EN PLACE DES CLASSES :

- COMPONENT : classe abstraite définissant les comportements communs aux objets : méthode(s) Opération(s)
- CONCRETECOMPONENT : concrétisation des classes : voiture du type CLIO, SCENIC...
- DECORATOR :
 - Cette classe encapsule (symbole UML de COMPOSITION) une référence vers un objet de type COMPONENT : Component component avec component le nom de la référence vers le type Component
 - Délègue les méthodes Opération vers la classe mère : component.Operation() => exploitation du polymorphisme (voir flèches en pointillées)
- CONCRETE DECORATOR :
 - Délègue les méthodes Opération vers les classes de bases (voir flèches en pointillées)
 - Rajoute des champs spécifiques (addedStated)
 - Rajoute des comportements spécifiques (AddedBehavior)
 -

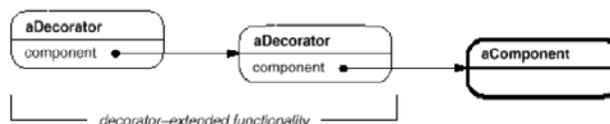
* MANIPULATION DES CLASSES :

Exemple de création d'objet :

```

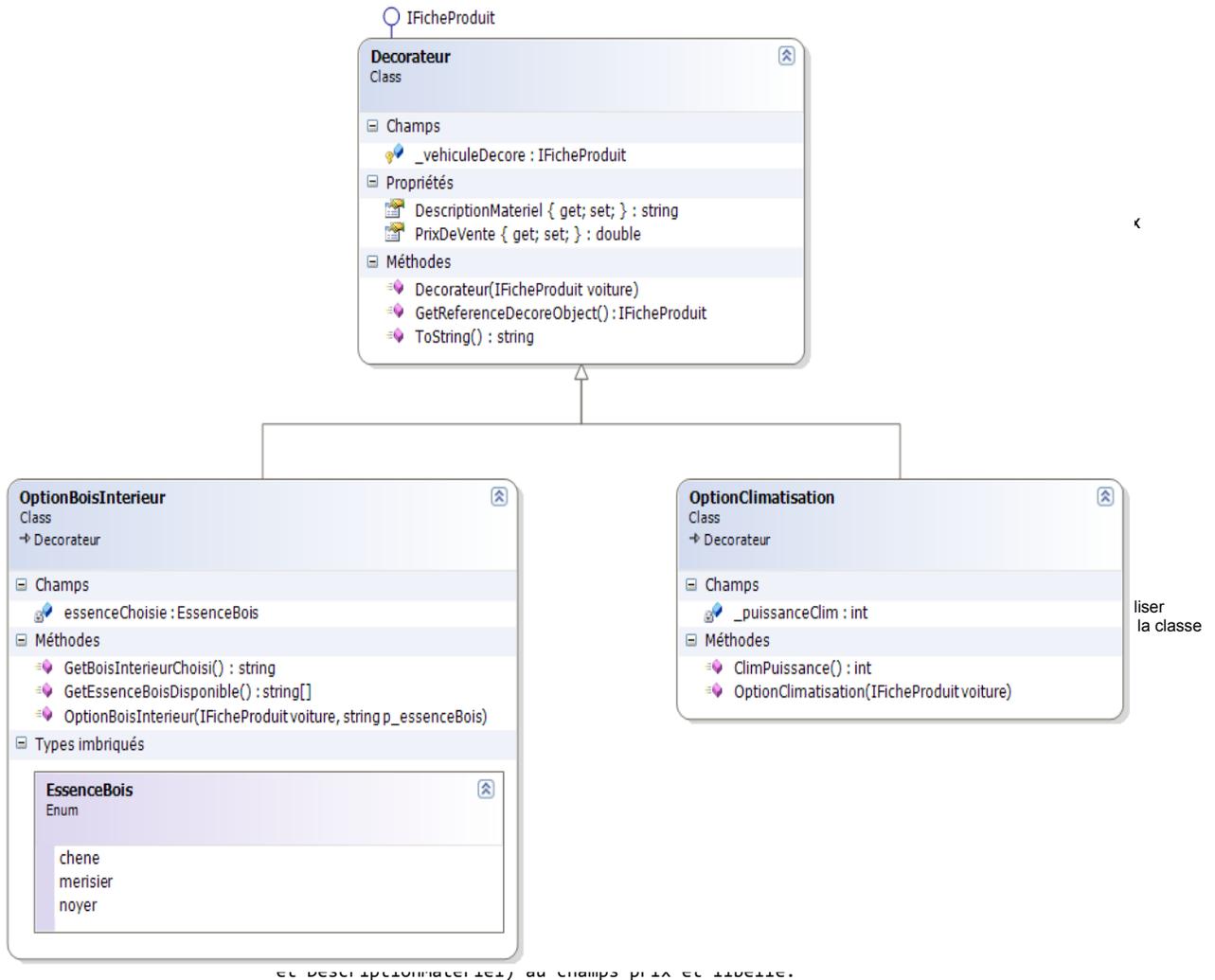
IFicheProduit uneVoitureClio = new Clio();
OptionBoisInterieur clioInterieurBois = new OptionBoisInterieur(uneVoitureClio, EssenceBoisDisponible[1]);
OptionClimatisation clioAvecClimetBoisInterieur = new OptionClimatisation(clioInterieurBois);
    
```

On notera le chaînage ainsi obtenu.



Travail à réaliser

Compléter l'exercice 2 de la préparation avec les classes suivantes :



Cette méthode est statique. Elle permet de retourner sous forme de tableau de string les essences de bois disponibles . En effet le type enum EssenceBois sera déclaré private, le rendant inaccessible depuis l'extérieur !

Vous utiliserez :

Pour convertir un string en type enum.

```
essenceChoisie = (EssenceBois) Enum.Parse(typeof(EssenceBois),p_essenceBois);
```

Pour convertir un type enum en tableau de chaîne

```
Enum.GetNames(typeof(EssenceBois))
```

Vous commencerez par OptionClimatisation

Vous créerez une clio. Afficher le prix et le libellé. Décorer cette clio d'une clim. Affichez le prix et le libellé.

Affichez la puissance de la climatisation.

Effectuer en pas à pas le programme et représenter sur papier les différents objets manipulés.

Compléter avec OptionBoisInterieur. Créer une scenic avec clim+BoisIntérieur par exemple.

Tester les conversions de type :