

Le langage C#

Aperçu du langage

Introduction

Comme nous l'avons déjà signalé, C# possède une syntaxe très proche de Java et donc du C. Cette partie du cours présente les principales caractéristiques du langage; et ne représente pas une description exhaustive de C#. A ce sujet, étant donné que C# est standardisé et public, on peut facilement trouver des documents basés sur la norme complète du langage C#.

Nous n'allons pas abandonner le rituel consistant à montrer le programme "Hello world!" en C#. Le voici:

```
using System;
class Hello
{
    static void Main() {
        Console.WriteLine("Hello, world");
    }
}
```

L'extension d'un programme C# est `.cs` (par exemple `hello.cs`) Le code source du programme peut être placé dans un ou plusieurs fichiers. Pour compiler ce programme, il faut taper la commande:

```
csc hello.cs
```

Ce qui produit un fichier `hello.exe` que l'on peut invoquer de la manière habituelle en tapant son nom sur la ligne de commande.

En regardant de plus près ce programme on peut faire plusieurs constatations:

- La première ligne `using System` indique une référence au namespace ou espace de nom (voir plus loin) `System` fourni par la librairie de classes CLI. Par exemple, ce namespace contient la classe `Console` appelée depuis le programme
- L'utilisation d'un namespace permet, entre autres, d'écrire `Console.WriteLine` au lieu de `System.Console.WriteLine`.
- La méthode `Main` est un membre de la classe `Hello`. Le spécificateur `static` indique qu'il s'agit d'une méthode de classe et non d'instance. Le point d'entrée pour l'exécution du programme est toujours la méthode statique `Main`.

Afin de pouvoir écrire des programmes simples utilisant la console, il est important non seulement de pouvoir afficher des informations, mais également de les saisir. Nous allons voir quelques exemples illustrant plusieurs manières d'utiliser les fonctionnalités d'entrée/sortie liées à la console.

On peut également accéder aux paramètres de la ligne de commande depuis un programme. C# propose ce mécanisme de manière analogue aux autres langages:

```

using System;
class Hello
{
    public static void Main (string[] args)
    {
        Console.WriteLine ("Hello, {0}!", args[0]);
        Console.WriteLine ("Bienvenu au club");
    }
}

```

Ainsi, si on tape `hello Ernest` le programme affiche:

```

Hello, Ernest!
Bienvenu au club

```

La classe `Console` possède une méthode `ReadLine` qui retourne une chaîne de caractères fournie par l'utilisateur:

```

using System;

class Hello
{
    public static void Main()
    {
        Console.Write("Entrez votre nom: ");
        string Nom = Console.ReadLine();
        Console.WriteLine("Bonjour "+ Nom);
    }
}

```

Voici un exemple d'exécution de ce programme:

```

Entrez votre nom: Arnold
Bonjour Arnold

```

Voici une autre variante n'utilisant pas de variable:

```

using System;

class Hello
{
    public static void Main()
    {
        Console.Write("Entrez votre nom: ");
        Console.WriteLine("Bonjour {0}", Console.ReadLine());
    }
}

```

Types

C# supporte deux sortes de types:

- Les **types valeur** qui incluent les types simples (char, int..), les types `enum` (énumérations) et les types `struct` (structures)
- Les **types référence** qui incluent les types `class` (classes), `interface` (interfaces), `delegate` (délégués) et `array` (tableaux)

Une variable de type valeur contient directement les données, alors qu'une variable de type référence ne contient que la référence à un objet (un peu comme le ferait un pointeur).

Voici un programme montrant la différence entre ces deux types:

```
using System;
class Class1
{
    public int Valeur = 0;
}

class Test
{
    static void Main() {
        int val1 = 0;
        int val2 = val1;
        val2 = 123;
        Class1 ref1 = new Class1();
        Class1 ref2 = ref1;
        ref2.Valeur = 123;
        Console.WriteLine("Valeurs: {0}, {1}", val1, val2);
        Console.WriteLine("Références: {0}, {1}", ref1.Valeur, ref2.Valeur);
    }
}
```

Ce programme affiche le résultat suivant:

```
Valeurs: 0, 123
Références: 123, 123
```

C# est un langage fortement typé, c'est-à-dire que toutes les opérations sur des variables sont effectuées en tenant compte de leur type, afin de maintenir l'intégrité des données affectées à ces variables.

Types prédéfinis

Les types référence prédéfinis sont les type `object` et `string`. Le type `object` est le type de base duquel tous les autres types héritent. Le type `string` permet de travailler avec des chaînes de caractères (Unicode) avec la particularité, comme Java, de ne pas pouvoir modifier la chaîne de caractères, une fois définie.

Les types valeur prédéfinis sont:

- Les types entiers signés et non signés
- Les types réels
- Le type `bool` (booléen). Nom long: `System.Boolean`

- Le type `char` (caractère)
- Le type `decimal` (pour représenter par exemple des données monétaires)

L'introduction des types `bool` et `string` permet de réduire les nombreuses erreurs de programmation habituelles en C et C++.

Voici un exemple courant d'erreur (parfois volontaire) signalée par le compilateur C#:

```
int i = .;
...;
if (i = 0) // Erreur signalée car doit s'écrire if (i==0)
...;
```

Le tableau suivant montre les types entiers, ainsi que leur taille et leur domaine de définition:

Type	Nom long	Taille (bits)	Domaine de définition
sbyte	System.SByte	8	-128 à 127
Byte	System.Byte	8	0 à 255
Short	System.Int16	16	-32768 à 32767
ushort	System.UInt16	16	0 à 65535
Int	System.Int32	32	-2147483649 à 2147483647
UInt	System.UInt32	32	0 à 4294967295
Long	System.Int64	64	-9223372036854775808 à 9223372036854775807
ulong	System.UInt64	64	0 à 18446744073709551615
Char	System.Char	16	0 à 65535

Remarque: la lettre 'u' devant un type signifie "non signé".

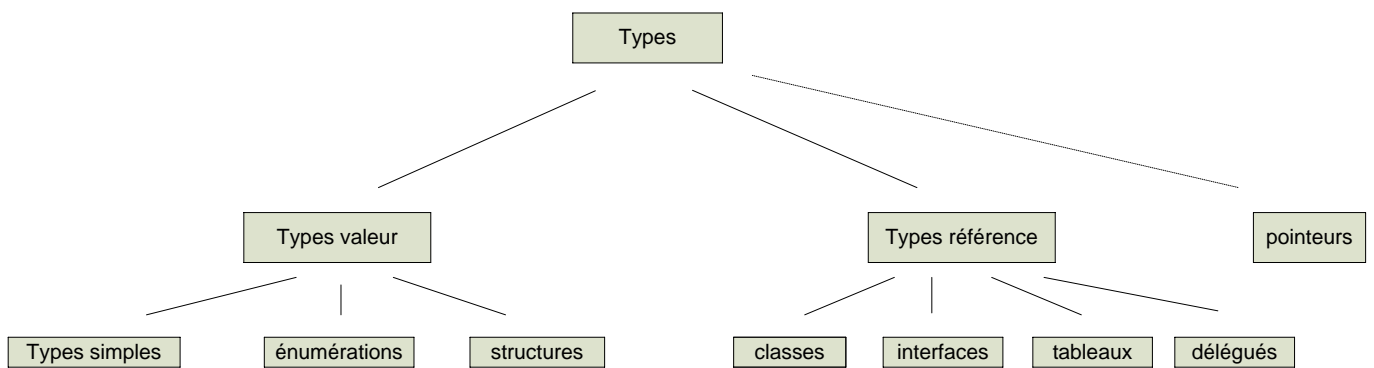
Le tableau suivant montre les types réels, leur taille, leur précision et leur domaine de définition:

Type	Nom long	Taille (bits)	Précision	Domaine de définition
Float	Sytem.Single	32	7 digits	$1.5 \cdot 10^{-45}$ à $3.4 \cdot 10^{38}$
double	Sytem.Double	64	15-16 digits	$5.0 \cdot 10^{-324}$ à $1.7 \cdot 10^{308}$
decimal	Sytem.Decimal	128	28-29 places déc.	$1.0 \cdot 10^{-28}$ à $7.9 \cdot 10^{28}$

Lors de l'utilisation de constantes littérales un suffixe peut être utilisé. Voici quelques exemples montrant comment spécifier des valeurs littérales:

Type	Description	Exemple
object	Type de base pour tous les autres types	object o = null;
string	Chaîne de caractères (Unicode)	string s = "bonjour";
sbyte	Entier signé 8-bits	sbyte val = 24;
short	Entier signé 16-bits	short val = 24;
int	Entier signé 32-bits	int val = 24;
Long	Entier signé 64-bits	long val1 = 24; long val2 = 54L;
Byte	Entier non signé 8-bits	byte val1 = 24;
ushort	Entier non signé 16-bits	ushort val1 = 24;
UInt	Entier non signé 32-bits	uint val1 = 24; uint val2 = 54U;
ulong	Entier non signé 64-bits	ulong val1 = 24; ulong val2 = 54U; ulong val3 = 68L; ulong val4 = 86UL;
float	Réel simple précision	float val = 1.45F;
double	Réel double précision	double val1 = 1.45; double val2 = 3.23D;
bool	Booléen	bool val1 = true; bool val2 = false;
char	Caractère (Unicode)	char val = 'w';
decimal	Precise decimal type with 28 significant digits	decimal val = 1.26M;

Le schéma suivant montre la classification des types en C#:



byte sbyte float
short ushort double
int uint decimal
long ulong

Remarques:

- Tous les types sont compatibles avec le type `object`, c'est-à-dire qu'ils peuvent être assignés à des variables de type `object`, et toutes les opérations disponibles pour le type `object` leur sont également applicables
- Comparons maintenant les types valeur et les types référence:

	Type valeur	Type référence
Une variable contient	une valeur	une référence
Stockée dans	le stack (<i>pile</i>)	le heap (<i>tas</i>)
Initialisation	0, false, '\0'	null
Assignation	copie de la valeur	copie de la référence
Exemple	<code>int i = 11;</code> <code>int j = i;</code>	<code>string s1 = "Salut"</code> <code>string s2 = s1;</code>

The diagram shows two scenarios. On the left, representing value types, two variables 'i' and 'j' are shown as boxes, each containing the value '11'. On the right, representing reference types, two variables 's1' and 's2' are shown as boxes. Both 's1' and 's2' have arrows pointing to a single box labeled 'Salut', indicating that both variables point to the same memory location containing the string 'Salut'.

Conversion de types

Les conversions de types sont monnaie courante en programmation. Prenons l'exemple suivant:

```
byte val1 = 12;
byte val2 = 20;
byte total;
total = val1 + val2;
Console.WriteLine (total);
```

Sa compilation provoque un message du genre: "Cannot implicitly convert type 'int' to 'byte' ". En effet, le résultat du calcul est un `int` et non un `byte`. Pour placer le résultat dans une variable de type `byte` il faut le reconvertir en `byte`.

Une conversion de type peut se faire explicitement ou implicitement.

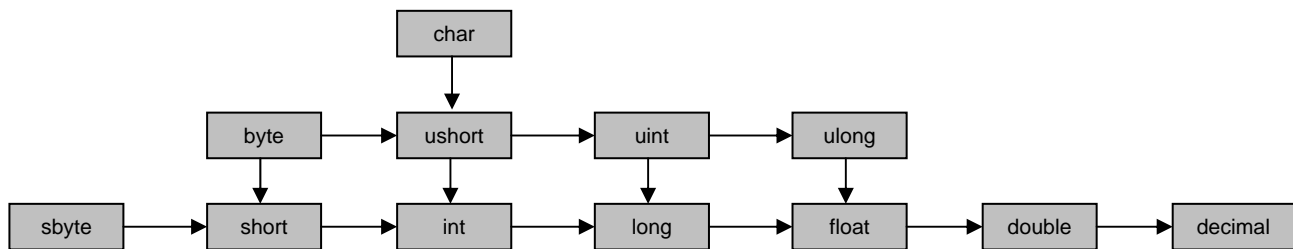
Conversions implicites

La conversion entre deux types peut s'effectuer de manière automatique (implicite) seulement si elle n'affecte pas la valeur convertie. Reprenons l'exemple précédent, mais avec la variable `total` de type `long`:

```
byte val1 = 12;
byte val2 = 20;
long total;
total = val1 + val2;
Console.WriteLine (total);
```

Cette fois le compilateur ne génère aucun message d'erreur.

Le schéma suivant donne les diverses possibilités de conversions implicites prises en charge par C#:



Conversions explicites

Pour beaucoup de conversions qui ne peuvent se faire implicitement, on a la possibilité d'utiliser le transtypage, c'est-à-dire une conversion explicite. En voici quelques exemples:

- de `int` en `short`
- de `float` en `int`
- d'un type numérique en `char`
- etc

Il convient d'être prudent avec l'utilisation du transtypage, car cette opération force le compilateur à effectuer une conversion d'où un risque d'altération des données. Voici un exemple typique de transtypage:

```
long tot = 20000;  
int k = (int)tot;
```

La conversion de `long` en `int` n'affecte pas la valeur 20000, car elle peut être contenue dans une variable de type `int`. En revanche le code qui suit est potentiellement dangereux:

```
long tot = 3000000000;  
int k = (int)tot;
```

car la variable `k` contient la valeur -1294967296, ce qui est manifestement faux et résulte de la troncature de 8 bytes à 4 bytes avec "mauvaise" prise en compte du bit de signe.

Une conversion courante en programmation concerne les nombres et les chaînes de caractères. Le fait que la classe `Object` implémente une méthode `ToString`, qui a été remplacée dans tous les type .NET prédéfinis, permet d'obtenir une représentation littérale de l'objet:

```
int k = 12;  
string s = k.ToString();
```

Pour effectuer la transformation inverse, c'est-à-dire passer d'une chaîne de caractères à un nombre il est possible d'utiliser la méthode `Parse`, offerte par tous les types prédéfinis:

```
string s = "120";
int k = int.Parse(s);
Console.WriteLine (k);
```

Il faut cependant remarquer que `Parse()` peut provoquer une erreur, et lever une exception qu'il faut traiter, si la conversion n'est pas faisable.

Boxing (emboîtement) et unboxing (désempoîtement)

Comme nous l'avons vu, tous les types, même les types simples prédéfinis, dérivent du type `Object`. Cela permet de travailler avec des valeurs littérales comme s'il s'agissait d'objets:

```
string s = 12.ToString();
```

Le **boxing** peut être vu comme un transtypage d'un type valeur à un type référence. Le runtime crée un "boîte" temporaire de type référence pour l'objet sur le tas (heap). Dans l'exemple précédent le boxing est effectué de manière implicite, mais on peut également le faire de façon explicite comme dans:

```
int k = 30;
object obj = k;           // boxing en int
```

Le unboxing concerne l'opération inverse dans laquelle un type référence est transtypé en type valeur.

```
int k = 30;
object obj = k;           // boxing d'un int en objet
int j = (int)obj;         // unboxing pour revenir à un int
```

Enumérations

Une énumération n'est rien d'autre qu'un type entier défini par l'utilisateur. En fait, pour plus de facilité d'utilisation, on travaille avec des noms qui, de manière implicite ou explicite sont associés à des valeurs numériques. Les avantages liés à l'utilisation des énumérations sont multiples:

- maintenance du code plus aisée avec la sécurité d'affecter à une variable uniquement des valeurs légitimes
- code plus clair par l'utilisation de noms descriptifs

Voici comment déclarer une énumération:

```
enum Couleur { // valeurs: 0, 1, 2
    rouge,
    bleu,
    vert
}

enum Acces { // valeurs explicites
    personel = 1,
    groupe = 2,
    tous = 4
}

enum Acces2 : byte { // possibilité d'associer un type
    personel = 1,
    groupe = 2,
    tous = 4
}
```

Et comment l'utiliser:

```
Couleur c = Couleur.bleu;

Acces a = Acces.personel | Acces.groupe;

if ((Acces.personel & a) != 0)
    Console.WriteLine ("Accès accordé");
```

Opérations avec les énumérations:

Il est possible d'effectuer des opérations avec les énumérations:

Comparaison	if (c == Couleur.rouge) if (c > Couleur.rouge && c <= Couleur.vert)
+ -	c = c + 1;
++ --	c++;
&	if ((c & Couleur.rouge) == 0)
	c = c Couleur.bleu;
~	c = ~ Couleur.rouge;

Tableaux

C# traite les tableaux de manière nouvelle. Bien que la syntaxe ressemble à celle que l'on trouve dans d'autres langages, leur implémentation est bien différente. Lors de la déclaration d'un tableau une instance de la classe .NET `System.Array` est créée. Le compilateur traduit les opérations habituelles sur les tableaux en appels de méthodes de `System.Array`.

Voici comment déclarer un tableau d'entiers:

```
int[] entiers;
```

Pour initialiser un tableau de taille définie on peut utiliser le mot-clé `new`:

```
int[] entiers = new int[20];
```

En C# tous les tableaux utilisent une indexation partant de zéro, et l'accès aux éléments du tableau s'effectue de manière habituelle:

```
entiers[0] = 42;
```

Avec la déclaration précédente, le dernier élément de notre tableau est:

```
entiers[19]
```

C# propose également la possibilité d'initialiser le contenu d'un tableau à l'aide d'une liste de valeurs:

```
string[] tableau = {"premier", "deuxième", "troisième"};
```

Utilisation des tableaux

Comme les tableaux correspondent à une classe sous-jacente, ils ont leurs propres méthodes et leur utilisation est facilitée. Par exemple:

- la propriété `Length` permet de connaître la taille d'un tableau:

```
int taille = entiers.Length;
```

- la méthode `Sort` permet de trier les éléments d'un tableau s'ils sont d'un des types prédéfinis

```
Array.Sort(tableau);
```

- il est également possible de renverser l'ordre des éléments dans un tableau:

```
Array.Reverse(tableau);
```

Tableaux multidimensionnels

En C# on trouve deux sortes de tableaux multidimensionnels:

Tableaux rectangulaires

Dans un tableau rectangulaire, par exemple à deux dimensions, toutes les lignes ont le même nombre de colonnes. Il s'agit en fait du type de tableaux que l'on trouve dans la plupart des langages de programmation, appelé aussi matrice. Voici un exemple d'un tel tableau ayant 4 lignes de 2 colonnes chacune:

```
string[,] savants = { { "Georges", "Charpak" },  
                      { "Albert", "Einstein" },  
                      { "Tim", "Berners-Lee" },  
                      { "Isaac", "Newton" } };
```

Voici un exemple d'utilisation de boucles pour initialiser les éléments d'un tableau à deux dimensions:

```
double[,] matrice = new double[10,10];  
for (int i = 0; i < 10; i++) {  
    for (int j = 0; j < 10; j++)  
        matrice[i,j] = 10;  
}
```

Tableaux orthogonaux

Cette sorte de tableau multidimensionnel est aussi appelée tableau de tableaux (*jagged*). Ces tableaux sont plus souples d'utilisation que les tableaux rectangulaires, mais plus difficiles à instancier et à initialiser:

```
int[][] a = new int[3][];  
a[0] = new int[4];  
a[1] = new int[3];  
a[2] = new int[1];
```

Les itérations sur les éléments d'un tableau orthogonal demandent plus de travail que pour un tableau rectangulaire. En effet, pour chaque ligne, il faut utiliser la méthode `GetLength` pour savoir combien la ligne possède de colonnes sur lesquelles itérer.

Structures

A l'origine une structure est une entité qui regroupe des champs dont le type peut être différent. En C#, une structure ressemble à une classe, avec quelques différences.

Déclaration de structure

Voici un exemple de déclaration d'une structure (mot réservé `struct`):

```

struct Point {
    public int x, y;          // champs
    public Point (int x, int y) { // constructeur
        this.x = x;
        this.y = y;
    }
    public void MoveTo (int a, int b) { // méthodes
        x = a;
        y = b;
    }
}

```

Utilisation d'une structure

L'utilisation d'une structure ressemble à celle d'une classe:

```

Point p = new Point(3, 4); // le constructeur initialise l'objet sur la pile
p.MoveTo(10, 20);        // appel de méthode

```

Classes

La classe est le concept de base de la programmation orientée objet. Elle est présentée ici afin de la comparer à la structure. On trouvera dans la suite plus d'indications concernant les classes.

Déclaration d'une classe

Voici comment déclarer une classe en C#:

```

class Rectangle {
    Point origine;                // champs
    public int largeur, hauteur;
    public Rectangle() {          // constructeur
        origine = new Point(0,0);
        largeur = hauteur = 0;
    }
    public Rectangle (Point p, int l, int h) { // autre constructeur
        origine = p;
        largeur = l;
        hauteur = h;
    }
    public void MoveTo (Point p) { // méthode
        origine = p;
    }
}

```

Utilisation d'une classe

Un exemple d'utilisation de la classe Rectangle:

```

Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int surface = r.largeur * r.hauteur;
r.MoveTo(new Point(3, 3));

```

Différences entre structures et classes

Le tableau qui suit montre les principales différences entre les structures et les classes:

Structures	Classes
Type valeur (objets stockés sur la pile)	Type référence (objets stockés sur le tas)
Pas de support de l'héritage (mais compatible avec object)	Support de l'héritage (toutes les classes dérivent de object)
Destructeurs non autorisés	Peuvent avoir un destructeur

Espaces de déclaration

Dans cette section nous allons voir où peuvent être déclarés les objets (au sens large) d'une application.

Espace de déclaration	Entités que l'on peut y déclarer
Namespace	Classes, interfaces, structures, énumérations, délégués
Class, interface, struct	Champs, méthodes, propriétés, événements, indexeurs...
Enum	Constantes d'énumération
Bloc	Variables locales

Domaine de définition

Les contraintes suivantes doivent être prises en considération:

- Un nom ne peut être déclaré plus d'une fois dans un domaine de déclaration.
- les déclarations peuvent intervenir dans n'importe quel ordre, sauf, bien entendu, pour les variables locales qui doivent être déclarées avant d'être utilisées

Règles de visibilité

La visibilité indique depuis quelle portion de programme un objet (au sens large) est visible et donc depuis où il peut être référencé.

- Un nom est visible uniquement dans son espace de déclaration.
- les variables locales sont visibles uniquement depuis (à partir de) leur point de déclaration, c'est-à-dire après leur point de déclaration, mais dans leur espace de déclaration

- la visibilité d'un nom peut être restreinte à l'aide de modificateurs (private, protected, ...)

Chaînes de caractères

Voici quelques exemples de déclarations et d'utilisation des chaînes de caractères.

```
string s = "Bonjour";           // Bonjour
string s = "\"Bonjour\"";      // "Bonjour"
string s = "Bon\njour";        // saut de ligne ajouté
string s = @"Bonjour\n";      // Bonjour\n
```

Indications:

- Il n'y a pas de limite quant au nombre de caractères constituent une chaîne de caractères.
- le caractère @ devant une chaîne de caractères permet de définir un "verbatim string", c'est-à-dire une chaîne de caractères dont le contenu est pris tel quel par le compilateur. Ceci peut se révéler particulièrement avantageux dans certains cas:

```
string document = @"c:\mes documents\nouveau\exemple.doc";
```

La seule exception est le guillemet, car le compilateur doit reconnaître la fin de la chaîne de caractères:

```
string s = @""Salut""; // produit la chaîne "Salut"
```

Constantes

Les constantes doivent être déclarées à l'aide du mot réservé `const`, suivi d'un type. Une valeur doit leur être affectée au moment de la déclaration:

```
const int noteMax = 6;
const long vitesseLumiere = 300000; // km/s
const double racineDeux = 1.414;
```

Expressions et opérateurs

En programmation on utilise une expression dans le but d'effectuer une action et de retourner une valeur. Une expression est une séquence d'opérateurs et d'opérandes. Un **opérateur** est un symbole indiquant l'action à effectuer, alors qu'un **opérande** est la valeur sur laquelle l'opération est effectuée.

Opérateurs courants	Exemples
---------------------	----------

Incrémentation/décrémentation	++ --
Arithmétiques	* / % + -

Relationnels	< > <= >= == !=
Conditionnels	&& ?:
Affectation	= *= /= %= += -= <<= >>= &= ^= =

Priorité des opérateurs

C# dispose quasiment des mêmes opérateurs que C. Voici les priorités des différents opérateurs

Groupe	Opérateurs
Primaire	(x) x.y f(x) a[x] x++ x- new typeof sizeof checked unchecked
Unaire	+ - ~ ! ++x --x (T)x (transtypage)
Multiplication/division	* / %
Addition/soustraction	+ -
Opérateurs de décalage binaires	<< >>
Relationnel	< > <= >= is as
Comparaison	== !=
ET binaire	&
XOR binaire	^
OU binaire	
ET booléen	&&
OU booléen	
Opérateur ternaire	? :
Affectation	= += -= *= /= %= &= ^= <<= >>= >>>=

Opérateurs et chaînes de caractères

Il est possible d'utiliser les opérateurs == et + avec les chaînes de caractères:

```
string a = "très";
string b = " bien";
string c = a + b; // très bien

if ("rouge" == "rouges")...
```

Instructions conditionnelles

Les instructions conditionnelles sont:

- l'instruction **if ... else**
- l'instruction **switch**

Une instruction conditionnelle permet de contrôler le flux d'un programme en fonction de la valeur d'une expression booléenne.

if ... else

Une instruction conditionnelle de type `if` possède deux syntaxes selon que la partie `else` est spécifiée ou non. De plus, conformément au théorème de structure, les instructions conditionnelles peuvent être imbriquées. En voici quelques exemples:

```
// des taxes sont calculées uniquement si les ventes excèdent 2000
if (ventes > 2000) {
    taxes = 0.072 * ventes;
}
```

```
if (ventes > 2000) {
    taxes = 0.072 * ventes; // exécuté si la condition est vraie
}
else {
    taxes = 0; // exécuté si la condition est fausse
}
```

```
if (ventes > 2000) {
    taxes = 0.072 * ventes;
}
else
    if (ventes > 1000) {
        taxes = 0.04 * ventes;
    }
    else {
        taxes = 0;
    }
}
```

Une alternative parfois intéressante, car concise, est l'opérateur ternaire `?:`:

```
taxes = (ventes > 2000) ? (ventes * 0.072) : 0;
```

switch

L'instruction `switch` peut être vue comme une instruction conditionnelle à plusieurs embranchements, en fonction de diverses valeurs que peut prendre l'expression testée.

```
switch (mois) {
    case 1 :
        jours = 31;
        break;
    case 2 :
        jours = 28;
        break;
    case 3 :
        jours = 31;
        break;
    case 4 :
        jours = 30;
        break;
    case 5 :
        jours = 31;
        break;
    ... etc
}
```

```
}
```

Dans ce cas, l'expression (`mois`) est évaluée. Si cette valeur est égale à une des constantes de `case`, l'exécution du programme se poursuit avec l'instruction qui suit le `case` correspondant. Si la valeur ne correspond à aucune des constantes proposées, l'exécution du programme se poursuit après l'instruction `switch`.

Il est également possible de spécifier une clause `default`. Si la valeur ne correspond à aucune des constantes proposées, l'exécution du programme se poursuit avec l'instruction qui suit `default` :

```
switch (mois) {
  case 1 :
    jours = 31;
    break;
  case 2 :
    jours = 28;
    break;
  case 3 :
    jours = 31;
    break;
  case 4 :
    jours = 30;
    break;
  case 5 :
    jours = 31;
    break;
  ... etc
  default :
    // numéro de mois incorrect
    break;
}
```

Il est obligatoire d'inclure l'instruction `break` à la fin de chaque bloc (contrairement au C et C++)

Il est toutefois possible de regrouper les traitements comme dans:

```
switch (mois) {
  case 1 :
  case 3 :
  case 5 :
  case 7 :
  case 8 :
  case 10 :
  case 12 :
    jours = 31;
    break;
  case 2 :
    jours = 28;
    break;
  default :
    jours = 30;
    break;
}
```

Instructions itératives (boucles)

En C# il existe trois instructions permettant d'effectuer des boucles:

- l'instruction `for`

- l'instruction `while`
- l'instruction `do`

Nous faisons ici uniquement un bref rappel. La syntaxe et l'utilisation est la même qu'en C, C++, Java.

Instruction for

La boucle `for` est d'une utilisation très souple, contrairement à l'équivalent en Pascal. Le plus souvent une boucle `for` est utilisée dans les situations où l'on connaît à l'avance le nombre d'itérations à effectuer.

La syntaxe est la suivante:

```
for (initialisation; condition; iteration) {
    instructions;
}
```

La condition est évaluée et, tant qu'elle est vraie, le bloc d'instructions constituant le corps de la boucle est exécuté. La condition est évaluée **avant** l'exécution des instructions. Les exemples qui suivent illustrent diverses utilisations d'une boucle `for`:

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine ("i = {0}", i);
}
```

```
for (int k = 200; k > 0; k -= 10) {
    Console.WriteLine ("k = {0}", k);
}
```

Le compteur de boucle peut être incrémenté ou décrémenté; il n'est pas forcément de type entier.

La partie initialisation et itération d'une boucle `for` peut contenir plus d'une déclaration de variable locale:

```
for (int i = 0, j = 100; i > 100; i++, j--) {
    Console.WriteLine ("{0}, {1}", i, j);
}
```

Les résultats produits par l'exécution de cette boucle sont:

```
0, 100
1, 99
2, 98
...
99, 1
```

Instruction while

Comme pour la boucle `for`, la boucle `while` est une boucle dans laquelle le test est effectué avant d'exécuter les instructions de la boucle.

La syntaxe est la suivante:

```
while (condition) {
    instructions;
```

```
}
```

La boucle `while` est souvent utilisée lorsque l'on ne connaît pas à l'avance le nombre d'itérations à effectuer:

```
bool continuer;  
...  
while (continuer) {  
    string res = LireElement();  
    // autres instructions  
    continuer = res != "fin";  
}
```

Instruction do

Contrairement aux instructions `for` et `while`, dans le cas de l'instruction `do` les instructions du corps de la boucle sont d'abord exécutées, ensuite seulement la condition de continuation de la boucle est évaluée.

La syntaxe est la suivante:

```
do {  
    instructions;  
} while (condition);
```

Le fait que le bloc d'instruction est de toute manière exécuté au moins une fois est important. Le programmeur doit prendre garde à cette particularité; par exemple lorsque la condition est fausse dès le départ ne doit pas poser de problème pour l'exécution des instructions de la boucle.

```
int somme = 0;  
int i = 0;  
do {  
    somme += i;  
} while (i <= 20);  
Console.WriteLine ("Somme des 20 premiers entiers: {0}, somme);
```

Instruction break et continue

Pour les trois types de boucles il est possible d'utiliser les instructions `break` et `continue`.

Lorsque l'instruction `continue` est rencontrée dans le corps d'une boucle, l'exécution se poursuit directement à la prochaine itération en ignorant complètement le reste des instructions du corps de la boucle.

Lorsque l'instruction `break` est rencontrée dans le corps de la boucle, l'exécution de la boucle est terminée et l'exécution se poursuit avec l'instruction qui suit la boucle.

Lorsque plusieurs instructions itératives sont imbriquées, l'effet des éventuelles instructions `break` et `continue` concerne uniquement la boucle dans laquelle elles apparaissent.

Les objets en C#

Dans cette partie nous allons aborder les concepts généraux de la programmation orientée objet, à savoir les notions de classes, d'objets et de méthodes.

Classes et objets

De manière simple on peut essayer de définir une classe et un objet. Les deux notions sont bien entendu intimement liées.

Classe

C'est un modèle à partir duquel on peut créer des objets. Une classe définit les caractéristiques d'un objet:

- les types de données que l'objet contient
- les méthodes décrivant le comportement de l'objet

Ces caractéristiques spécifient comment les autres objets peuvent accéder et travailler avec les données de l'objet considéré.

Objet

Un objet est une instance d'une classe. Il est créé à partir du "modèle" qu'est la classe.

Définition d'une classe et création d'un objet

La définition d'une classe suit la syntaxe:

```
[attributs] [modificateurs d'accès] class identificateur { corps }
```

(nous reviendrons plus loin sur les attributs et les modificateurs d'accès)

Définissons une nouvelle classe, `Client`, contenant trois informations (ou champs, ou encore membres): le nom, le revenu et le no. de client:

```
class Client {  
    public string nom;  
    public decimal revenu;  
    public uint clientNo;  
}
```

Bien que la classe soit définie, il n'existe encore aucun objet `Client`. La définition d'une classe correspond à créer un nouveau type dans une application. Afin d'utiliser la classe ainsi créée il faut instancier un objet de ce type, à l'aide du mot-clé `new`. La syntaxe est:

```
<classe> <objet> = new <classe>
```

Par exemple:

```
Client unClient = new Client();
```

Accès aux variables (champs) d'une classe

Après avoir instancié un objet, l'accès aux champs d'une classe est effectué en utilisant la notation pointée:

```
unClient.nom = "Pierre Platte";
```

Remarques:

- rappelons qu'une classe est un type référence
- il est fortement conseillé de suivre la casse Pascal pour les noms de classes (par exemple `MaPremiereClasse`). Chaque mot qui compose le nom commence par une majuscule
- lors de la création d'un objet un emplacement mémoire est alloué. A l'instar de Java, il n'est pas de la responsabilité du programmeur de désallouer la mémoire utilisée par un objet. Le garbage collector (ou ramasse-miettes) s'en charge automatiquement. Dès qu'un objet n'est plus utilisé il libère la mémoire qu'il occupe. Le moment auquel cette libération intervient n'est pas prédictible. Il existe toutefois une possibilité de libérer explicitement la mémoire, mais nous ne l'aborderons pas ici.

Namespaces (espaces de noms)

Les namespaces servent à organiser les classes dans une hiérarchie logique, un peu à la manière des packages de Java, mais de manière plus souple.

Les namespaces ne sont pas seulement utiles dans une optique d'organisation interne, mais aussi dans le but d'éviter des collisions de noms. Il est fortement probable que beaucoup de sociétés et de développeurs ont défini une classe qui s'appelle `Client`. L'utilisation de namespaces permet de lever l'ambiguïté des noms de classes.

On peut donc dire qu'un `namespace` est un *système d'organisation permettant d'identifier des groupes de classes*.

Remarques:

- il est recommandé d'utiliser la casse Pascal pour les noms de namespaces
- dans un environnement de développement professionnel il est également conseillé de créer des namespaces à double niveau. Un premier niveau indiquant le nom de la société est suivi d'un second niveau spécifiant par exemple le nom du département:

```
namespace LesComptoirs.Ventes {
    public class Client(){
    }
}
```

cette notation est équivalente à la suivante, qui utilise un format imbriqué:

```
namespace LesComptoirs {
    namespace Ventes {
        public class Client(){
        }
    }
}
```

et dans les deux cas la classe `Client` peut être référencée par:

```
LesComptoirs.Ventes.Client()
```

ce qui constitue le *fully qualified name* de la classe `Client`

Les namespaces du framework .NET

Le framework .NET est constitué d'un grand nombre de namespaces, dont le plus important est `System`. Ce namespace contient des classes que la plupart des applications utilisent pour interagir avec le système d'exploitation.

Par exemple, le namespace `System` contient la classe `Console` qui fournit diverses méthodes, dont `WriteLine`. On peut ainsi écrire:

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

Il n'y a pas de limite au nombre de niveaux d'un namespace. De ce fait, l'écriture de programmes peut devenir fastidieuse et peu claire. C'est pour pallier à cet inconvénient que la directive `using` a été introduite.

Normalement au début des fichiers de programmes on trouve une liste des namespaces utilisés dans le fichier, préfixés par `using`.

```
using System;
using LesComptoirs.Ventes;
...
Console.WriteLine ("Bonjour");
Client unClient = new Client();
```

Accessibilité et domaine de visibilité (scope)

Le concept de domaine de visibilité a pris toute son importance surtout depuis l'avènement de la programmation structurée et des langages de programmation modernes. Pour des raisons de sécurité, d'organisation et de maintenance il est important que les divers objets d'un programme n'aient pas tous la même accessibilité. Certains doivent être masqués, d'autres publics.

Le **domaine de visibilité** d'un objet (au sens large) est *la région du code d'une application depuis laquelle cet objet peut être référencé*.

En C# les modificateurs d'accès sont utilisés dans le but de contrôler leur domaine de visibilité. Le tableau suivant présente les modificateurs d'accès disponibles pour les membres d'une classe ainsi que leur signification:

Déclaration	Définition
Public	Accès illimité. N'importe quelle autre classe peut accéder à un membre public
Private	Accès limité à la classe contenant. Seule la classe contenant le membre peut accéder au membre private.
Internal	Accès limité à cet assemblage (programme). Les classes se trouvant dans le même assemblage peuvent accéder au membre.
Protected	Accès limité à la classe contenant et aux classes dérivées de cette classe.
protected internal	Accès limité à la classe contenant, aux classes dérivées et aux classes se trouvant dans le même assemblage.

Un **assemblage (assembly)** est *l'ensemble des fichiers constituant un programme*.

Règles importantes:

- les namespaces sont toujours `public` (implicitement).
- les classes sont toujours `public` (implicitement).
- les membres sont `private` par défaut.
- un seul modificateur d'accès peut être défini pour un membre de classe (`protected internal` est considéré comme un unique modificateur).
- le domaine de visibilité d'un membre n'est jamais plus grand que celui du type qui le contient.

Remarques:

- `public` doit donc être réservé aux entités que les utilisateurs d'une classe ont besoin de voir.
- en limitant le nombre d'entités `public` on diminue la complexité des classes du point de vue de l'utilisateur. Cela rend la maintenance plus aisée.

Afin de mieux comprendre le concept d'accessibilité, considérons l'exemple suivant:

```
using System;

namespace CoursCSharp.Exemple {
    class ClassMain {

        public class Troll {
            public int age;
            private string couleur;
        }
        static void Main (string[] args) {
            Troll tiTroll = new Troll();
            tiTroll.age = 100;
            tiTroll.couleur = "rouge";    // erreur car couleur n'est pas accessible
        }
    }
}
```

Méthodes

Une **méthode** est un membre d'une classe qui est utilisé pour définir une action pouvant être accomplie par cet objet ou classe.

Syntaxe de déclaration d'une méthode:

```
[attributs] [modificateurs d'accès] type-retour nom ([liste de paramètres]) { corps }
```

Règles et recommandations:

- le type de retour d'une méthode doit toujours être spécifié. Si une méthode ne retourne rien, ce type sera `void`.
- même si une méthode ne comporte pas de paramètres son nom doit être suivi de parenthèses vides `()`.

- bien entendu lorsqu'une méthode est invoquée, il faut respecter le type de la valeur retournée, ainsi que le nombre, l'ordre et le type des paramètres.
- le nom d'une méthode devrait représenter une action. Par exemple: DiminuerAge, AlimenterTroll.
- il est conseillé que les noms des méthodes suivent une casse Pascal.

Exemple:

```
class Troll {
    private int poids;

    public bool TesterPoidsNormal () {
        if ((poids < 20) || (poids > 40)) {
            return false;
        }
        return true
    }
    public void Manger () { }
    public int GetPoids () {
        return poids;
    }
}
```

Création d'un objet Troll:

```
Troll grosTroll = new Troll();
```

Test de son poids:

```
if (grosTroll.TesterPoidsNormal () == false) {
    Console.WriteLine ("Le troll n'a pas un poids idéal");
}
```

La méthode Manger ne retourne aucune valeur. Elle pourrait être utilisée ainsi:

```
grosTroll.Manger();
```

Le mot-clé this

Le mot-clé `this` est utilisé pour indiquer l'instance courante de l'objet. Nous aurions pu écrire la méthode `GetPoids` de la manière suivante (équivalente à la définition précédente):

```
public int GetPoids () {
    return this.poids;
}
```

Passage de paramètres

Le passage de paramètres est le mécanisme permettant d'échanger de l'information entre une méthode et la portion de programme appelante.

Passage de paramètres par valeur

Lorsqu'une variable de type valeur est passée à une méthode, celle-ci reçoit une copie de la valeur de cette variable:

```

class Troll {
    private int poids;
    public void SetPoids ( int nouveauPoids) {
        poids = nouveauPoids;
    }
}
...
Troll pasTroll = new Troll();

int lePoids = 45;
pasTroll.SetPoids (lePoids); // appel de la méthode

```

Lors de l'appel la valeur de `lePoids` est copiée dans le paramètre `nouveauPoids` (qui est en fait considérée comme une variable locale à la méthode).

Voici un second exemple:

```

class Champ {
    public int CalculerSurface (int largeur, int longueur) {
        return largeur * longueur;
    }
}
...

Champ monChamp = new Champ();
int surface = monChamp.CalculerSurface (40, 60);

```

Passage de paramètres par référence

Nous avons vu qu'une méthode retourne une valeur unique. Dans certaines situations on aimerait qu'une méthode retourne plusieurs valeurs. Il est alors possible de passer à la méthode une référence à la variable sur laquelle la méthode va travailler. Toute modification de l'objet passé à la méthode sera répercutée sur l'objet à la fin de l'exécution de la méthode. Pour indiquer un passage de paramètre par référence, le mot-clé `ref` doit être spécifié.

Reprenons notre classe `Client`:

```

class Client {
    public string nom = "Pierre";
    public decimal revenu = 1000D;
    public uint clientNo = 22231;
    public void GetClient (ref string leNom,
                           ref decimal leRevenu,
                           ref unit leNo) {
        leNom = nom;
        leRevenu = revenu;
        leNo = clientNo;
    }
}
...
Client unClient = new Client();
string unNom = null;
decimal unRevenu = 0D;
uint unNo = 0;
...

unClient.GetClient (ref unNom, ref unRevenu, ref unNo);

```

On remarquera que le mot-clé `ref` doit également être spécifié lors de l'appel de la méthode.

Remarques:

- C# oblige que toutes les variables soient initialisées avant d'être passées comme paramètres à une méthode. Même si, comme dans notre exemple, la méthode initialise ces variables.
- C# dispose du mot-clé `out` permettant de spécifier au compilateur qu'un paramètre est initialisé par une méthode. Dans ce cas il est possible de passer, par référence, une variable non initialisée:

```
class Client {
    public string nom = "Pierre";
    public decimal revenu = 1000D;
    public uint clientNo = 22231;
    public void GetClient (out string leNom,
                           out decimal leRevenu,
                           out unit leNo) {
        leNom = nom;
        leRevenu = revenu;
        leNo = clientNo;
    }
}
...
Client unClient = new Client();
string unNom;
decimal unRevenu;
uint unNo;
...

unClient.GetClient (out unNom, out unRevenu, out unNo);
```

Passage d'un type référence comme paramètre

Lorsqu'une variable de type référence (par exemple un objet) est passée comme paramètre à une méthode, celle-ci peut altérer la valeur de cette variable. L'exemple suivant met en évidence cette caractéristique:

```
class MainClass {
    static void Main (string[] args) {
        Zoo monZoo = new Zoo();
        Troll tiTroll = new Troll();

        monZoo.AjouterTroll (tiTroll);
        // ici tiTroll.emplacement vaut "Cage no. 3"
    }
}

class Troll {
    public string emplacement;
}

class Zoo {
    public void AjouterTroll (Troll nouveauTroll) {
        nouveauTroll.emplacement = "Cage no. 3";
    }
}
}
```

Surcharge de méthode

La **surcharge de méthode** (overload) est *une particularité permettant, dans une même classe, de définir plusieurs méthodes ayant le même nom, mais une signature différente*. Par signature on entend l'ensemble constitué par le nom de la méthode et ses paramètres.

La classe Date comporte plusieurs méthodes surchargées:

```
class Date {
    public void Ajouter (int jours);
    public void Ajouter (int jours, int mois, int annees);
    public void Ajouter (double temps);
}
```

L'utilisation de la surcharge de méthodes présente plusieurs avantages:

- l'utilisation du même nom (donc même signification) pour plusieurs "variantes" d'actions
- facilité pour ajouter de nouvelles fonctionnalités en changeant uniquement la syntaxe d'appel d'une méthode
- fournir plusieurs constructeurs pour une classe donnée (voir plus loin)

Constructeurs

Un **constructeur** est *une méthode particulière utilisée pour initialiser un objet*.

Chaque classe intègre, de manière implicite ou explicite, un constructeur d'instance. Cette méthode est automatiquement appelée lorsqu'une nouvelle instance de classe est créée. Un constructeur porte le même nom que la classe pour et dans laquelle il est défini.

```
class Boisson {
    public Boisson() {
        Console.WriteLine ("Constructeur boisson");
    }
}
```

Voici comment le constructeur est utilisé:

```
Boisson sirop = new Boisson();
Console.WriteLine ("Sirop créé");
```

on obtient le résultat suivant:

```
Constructeur boisson
Sirop créé
```

Il n'est pas obligatoire de spécifier un constructeur. Dans ce cas C# en fournit un par défaut (sans paramètres):

```
class Boisson {
    private string marque;
}
```

est équivalent à:

```
class Boisson {
    private string marque;
    public Boisson() {
    }
}
```

Un constructeur peut bien entendu avoir des paramètres. Si un tel constructeur est spécifié, alors aucun constructeur par défaut n'est automatiquement créé.

```
class Boisson {
    private string marque;
    public Boisson(string laMarque) {
        marque = laMarque;
    }
}
```

De cette manière l'utilisateur de la classe `Boisson` est obligé d'indiquer la marque de la boisson lorsqu'il crée une boisson:

```
Boisson sirop = new Boisson("grenadine");
```

Remarques:

- on peut donc constater qu'un constructeur porte le nom de la classe et ne retourne rien, même pas le type `void`
- Au niveau des constructeurs d'une classe, on peut donc se trouver dans une des situations suivantes:
 - aucun constructeur n'est spécifié. Le compilateur fournit automatiquement un constructeur par défaut, sans paramètres
 - un seul constructeur est spécifié. Dans ce cas, c'est ce constructeur qui sera appelé. Le compilateur ne fournit pas de constructeur par défaut
 - plusieurs constructeurs sont spécifiés. Tous les constructeurs diffèrent par leur signature. On est dans le cas d'un constructeur qui est une méthode surchargée un certain nombre de fois. Le compilateur ne fournit pas de constructeur par défaut

A la fin de l'exécution d'un constructeur, tous les champs d'instance (donc les champs non statiques) doivent avoir été initialisés, que ce soit par le constructeur ou lors de leur déclaration.

```
class Boisson {
    public string marque;
    public Boisson() {
        marque = "no name";
    }
}
```

Dans notre exemple, pour toutes les instances de la classe `Boisson` le champ `marque` est initialisé à "no name". On aurait donc très bien pu écrire:

```
class Boisson {
    public string marque = "no name";
    public Boisson() {
        // autres instructions
    }
}
```

Il est possible d'appeler un constructeur depuis un second constructeur à l'aide du mot réservé `this` comme dans l'exemple qui suit. `this(5)` appelle le second constructeur, avant que le code du premier constructeur ne soit exécuté (vide dans notre cas).

```
class MaClasse {
    public int x;
    public MaClasse() : this(5) {}
}
```

```

    public MaClasse (int v) {
        x = v;
    }
}

MaClasse c1 = new MaClasse ();
MaClasse c2 = new MaClasse (10);
Console.WriteLine (c1.x);           // écrit 5
Console.WriteLine (c2.x);           // écrit 10

```

Remarque:

On peut spécifier n'importe quel modificateur d'accès pour les constructeurs. Dans certains cas le constructeur peut être `private` afin d'interdire que la classe soit instanciée. Cela peut être souhaitable pour certaines classes d'utilitaires contenant uniquement des membres statiques, par exemple la classe `System.Math`.

Membres statiques

Les champs, les propriétés, les méthodes et les événements d'une classe peuvent être déclarés avec l'attribut `static`. Ces membres statiques appartiennent à la classe et non aux objets instanciés à partir de la classe. On parle également de *membres de classe*, par opposition aux *membres d'instance*.

Les mêmes règles de visibilité sont appliquées aux membres statiques.

L'exemple suivant montre que l'on peut utiliser des membres statiques d'une classe sans qu'aucun objet de cette classe ne soit créé.

```

class Robot {
    public static string controle = "autonome";
}
...
Console.WriteLine ("Contrôle: {0}", Robot.controle);

```

Les méthodes peuvent aussi être statiques. De telles méthodes sont alors accessibles uniquement par la classe.

Considérons une classe `Etudiant` dans laquelle on désire stocker le nombre d'étudiants instanciés. On pourrait écrire:

```

class Etudiant {
    private int totalEtudiants;
    ...
    public void AfficherTotal() {
        Console.WriteLine ("Nombre d'étudiants: " + totalEtudiants);
    }

    public void InscrireEtudiant() {
        TotalEtudiants = TotalEtudiants + 1;
    }
    ...
}

```

Cette manière de faire pose deux problèmes:

- il y aura autant de fois l'information `totalEtudiants` qu'il y aura d'objets instanciés, ce qui , outre la redondance de l'information, introduit un gaspillage de place.
- il est dommage d'appeler la méthode `InscrireEtudiant` de chaque objet, uniquement pour tenir à jour le nombre total d'étudiants

Dans un premier temps on peut utiliser une variable de classe (statique) pour compter le nombre d'étudiants

```
class Etudiant {
    private static int totalEtudiants;
    ...
    public void AfficherTotal() {
        Console.WriteLine ("Nombre d'étudiants:  " + totalEtudiants);
    }

    public void InscrireEtudiant() {
        totalEtudiants = totalEtudiants + 1;
    }
    ...
}
```

Il est également possible d'implémenter une propriété statique permettant d'accéder à `totalEtudiants`:

```
class Etudiant {
    private static int totalEtudiants;
    ...
    public static int TotalEtudiants {
        get {
            return totalEtudiants;
        }
        set {
            totalEtudiants = value;
        }
    }

    public void InscrireEtudiant() {
        TotalEtudiants = TotalEtudiants + 1;    // avec 'T'
    }
    ...
}
```

La propriété statique peut être invoquée uniquement depuis la classe `Etudiant`:

```
Console.WriteLine ("Nombre d'étudiants:  " + Etudiant.TotalEtudiants);
```

Si on essaie d'invoquer une propriété statique sur un objet le compilateur génère une erreur, comme pour:

```
Etudiant et1 = new Etudiant();
Console.WriteLine ("Nombre d'étudiants:  " + et1.TotalEtudiants);
```

Afin d'améliorer encore notre exemple, il est possible de faire appel aux méthodes statiques:

```
class Etudiant {
    private static int totalEtudiants;
```

```

...
    public static int TotalEtudiants {
        get {
            return totalEtudiants;
        }
        set {
            totalEtudiants = value;
        }
    }

    public static void AfficherTotal() {
        Console.WriteLine ("Nombre d'étudiants:  " + TotalEtudiants);
    }

    public static void InscrireEtudiant() {
        TotalEtudiants = TotalEtudiants + 1;    // avec 'T'
    }
...
}

```

De cette manière toutes les informations et les traitements qui concerne uniquement la classe sont spécifiés comme statiques. Voici un exemple d'utilisation de la classe `Etudiant`:

```

Etudiant et1 = new Etudiant();
et1.nom = "Pierre",
Etudiant.InscrireEtudiant();

Etudiant et2 = new Etudiant();
Et2.nom = "Josette",
Etudiant.InscrireEtudiant();

Etudiant et3 = new Etudiant();
Et3.nom = "Astrid",
Etudiant.InscrireEtudiant();

Etudiant.AfficherTotal();    // affiche Nombre d'étudiants: 3

```

Constructeurs statiques

Un constructeur d'instance est utilisé afin d'initialiser un objet. On peut également spécifier un constructeur qui initialise une classe. On parle alors de **constructeur statique**.

Un constructeur statique:

- a le même nom que la classe
- n'a pas de paramètres
- est exécuté une seule fois
- peut coexister avec des constructeurs d'instances
- ne peut pas être appelé explicitement. Il est appelé une fois que la première instance de soit créée, ou avant qu'une des méthodes statiques ne soit invoquée.

Héritage

Lorsque l'on développe une application on est souvent amené à travailler avec des éléments similaires, mais pas identiques. En programmation objet le principe de l'héritage permet, dans un premier temps, de définir une classe, puis, dans un second temps, de dériver de cette classe initiale des classes plus spécifiques.

La classe initiale est appelée **classe de base**, alors que les classes construites à partir de cette classe de base sont appelées **classes dérivées**. Les classes dérivées héritent des propriétés et des méthodes de la classe de base.

L'héritage permet donc de spécifier des méthodes et des propriétés communes ou générales et de les réutiliser dans une classe dérivée. L'héritage présente donc un avantage en ce qui concerne la réutilisabilité du code.

Dans l'exemple qui suit on désire travailler avec des animaux et la première idée d'implémentation pourrait être de créer une classe pour chaque animal:

```
public class Chat {
    public bool EstEnerve;
    public void Mange();
    public void Dort();
}

public class Lion {
    public bool EstEnerve;
    public void MangeZebre();
    public void Dort();
    public void Rugit();
}

public class Elephant {
    public int ChargeMaximale;
    public bool EstEnerve;
    public void Dort();
    public void Mange();
}
```

Cette structure mène à une certaine duplication du code. En effet la méthode `Dort` est implémentée plus d'une fois. D'autre part, la méthode traduisant le fait de manger ont des noms différents

Il est plus judicieux de définir une structure mettant en œuvre l'héritage:

```
public class Animal {
    public bool EstEnerve;
    public void Mange() { };
    public void Dort() {
        Console.WriteLine ("en train de dormir...");
    };
}

public class Chat : Animal {
}

public class Lion : Animal {
    public void Rugit();
}
```

```
public class Elephant : Animal {
    public int ChargeMaximale;
}
```

Les méthodes `Dort` et `Mange` sont définies une seule fois dans la classe de base `Animal`. Les trois classes dérivées héritent de ces méthodes:

```
Elephant e = new Elephant();
e.Dort(); // affiche "en train de dormir..."
```

Une classe peut hériter de n'importe quelle classe qui n'est pas définie avec l'attribut `sealed` (*héritage interdit*), donc également d'une classe elle-même dérivée.

Reprenons notre exemple en introduisant une classe `Mammifere` dérivée de la classe `Animal`:

```
public class Animal {
    public bool EstEnerve;
    public void Mange() { };
    public void Dort() {
        Console.WriteLine ("en train de dormir...");
    };
}

public class Mammifere : Animal {
    public SousGroupe Sorte;
}

public class Chat : Mammifere {
}

public class Lion : Mammifere {
    public void Rugit();
}

public class Elephant : Mammifere {
    public int ChargeMaximale;
}
```

L'énumération `SousGroupe` serait définie comme:

```
public enum SousGroupe {
    Monotremes, // p.ex. ornithorynque, équidné
    Marsupiaux, // p.ex. kangourou, koala
    Placentaires // p.ex. chien, lion
}
```

On peut, par exemple, écrire:

```
Elephant e = new Elephant();
e.Dort();
e.Sorte = SousGroupe.Placentaires;
```

Appel d'un constructeur de la classe de base

Comme nous l'avons vu, lors de la création d'un objet, un constructeur d'instance est exécuté. Lors de la création d'un objet d'une classe dérivée:

- le constructeur d'instance de la classe de base est exécuté en premier lieu

- puis le constructeur d'instance de la classe dérivée est exécuté

```
public class Animal {
    public Animal() {
        Console.WriteLine ("Construction de Animal");
    };
}

public class Elephant : Animal {
    public Elephant() {
        Console.WriteLine ("Construction de Elephant");
    }
}
```

Lorsqu'un objet `Elephant` est créé:

```
Elephant e = new Elephant();
```

Les lignes suivantes sont affichées:

```
Construction de Animal
Construction de Elephant
```

L'ordre d'exécution des constructeurs suit l'ordre de la hiérarchie des classes.

Dans le cas où la classe de base possède un constructeur (autre que celui par défaut) que l'on veut appeler, il faut utiliser le mot-clé `base`. Si la classe de base `Animal` se présente sous la forme suivante:

```
public enum LeGenre {
    Male,
    Femelle
}

public class Animal {
    public Animal() {
        Console.WriteLine ("Construction de Animal");
    }

    public Animal (LeGenre genre) {
        if (genre == LeGenre.Femelle) {
            Console.WriteLine ("Femelle");
        }
        else {
            Console.WriteLine ("Male");
        }
    }
}
```

Voici comment appeler comment appeler le second constructeur de la classe `Animal` depuis la classe dérivée `Elephant`:

```
public class Elephant : Animal {
    public Elephant (LeGenre genre) : base (genre) {
        Console.WriteLine ("Elephant");
    }
}
```

Si on crée alors un objet de type `Elephant`:

```
Elephant e = new Elephant (LeGenre.Femelle);
```

on obtient:

```
Femelle  
Elephant
```