

1. C# et l'architecture .NET

1.1. Introduction

.NET tourne dans l'environnement Windows 98, 2000 ou XP. Pour écrire du code utilisant .NET, vous devez installer le SDK .NET téléchargeable sur le site Microsoft ou l'outil de développement Visual studio .NET. Pour exécuter votre application sur une plate-forme donnée, vous devez avoir installé sur cette plate-forme le runtime .NET connu aussi sous le nom de Common Language Runtime (CLR).

Avant d'être exécuté par le CLR, tout code source développé en C# doit être compilé dans un langage intermédiaire appelé MS-IL (Microsoft Intermediate Language). Ce code est à son tour compilé en temps réel par le CLR en code spécifique à une plate-forme donnée. Actuellement, .NET n'est disponible que pour Windows mais il existe un projet de création d'une implémentation Open Source. Vous pourrez trouver plus de documentations à l'adresse <http://www.go-mono.com>.

1.2. Premier programme en C#

```
1.     using System;
2.     namespace MyNameSpace
3.     {
4.         public class MyFirstApp
5.         {
6.             public static void Main()
7.             {
8.                 // Une simple ligne de commentaire
9.                 /* Plusieurs lignes
10.                de
11.                commentaire */
12.                Console.WriteLine("Bonjour à tous");
13.            }
14.        }
15.    }
```

En Visual studio 6.0, les habitués de la programmation en C++ ont souvent utilisés les MFC (Microsoft Foundation Class) offrant des possibilités rapides de développement pour les applications orientées Windows. En C#, on retrouve également des bibliothèques de classes de base en code IL (appelé aussi code managé). L'ensemble de ces bibliothèques se retrouvent dans ce que l'on appelle le .NET Framework.

C# est un langage orienté objet et de ce fait les programmes doivent être placés dans des classes. Nous retrouvons à la ligne 4 notre classe MyFirstApp. La déclaration de la classe s'effectue au moyen du mot clef *class*

A la ligne 1, nous déclarons que nous utilisons un espace de nom appelé System tandis qu'à la ligne 2, nous déclarons notre propre espace de nom MyNameSpace dans lequel nous déclarons notre classe. Les espaces de noms sont un moyen d'éviter les collisions de noms entre classes. Deux classes appartenant à des espaces de noms différents peuvent porter le même nom sans que cela ne pose problème. Un espace de noms n'est rien d'autre qu'un regroupement de types de données dont le nom sera automatiquement préfixé par celui de l'espace de noms. Il est possible d'imbriquer les espaces de nom.

A la ligne 12, nous appelons la méthode WriteLine qui appartient à la classe de base Console définie elle même dans l'espace de nom System (c'est dans cet espace de noms que résident les types .NET les plus souvent utilisés).

La syntaxe complète d'accès à la méthode WriteLine pourrait être:

```
System.Console.WriteLine("bonjour à tous");
```

Pour éviter la complexité d'écriture du code, on peut définir les espaces de noms avec lesquels on travaille pour ainsi limiter l'écriture à Console.WriteLine("bonjour à tous");

Lorsque nous écrivons un programme DOS en C++, nous devons définir une *fonction void main()* qui sera appelée lors de l'exécution de ce programme. Si c'est un programme Windows, la fonction doit s'appeler WinMain().

En C#, les exécutables (applications console, applications et services Windows) doivent avoir comme point d'entrée la méthode *Main()*. (Notons la majuscule pour le nom de la fonction). Cette méthode doit retourner soit un entier, soit rien se traduisant par le mot clef *void*.

La compilation sous Visual .NET peut s'effectuer en appuyant sur les touches Ctrl-Shift-B. Cette compilation produira un exécutable dont l'exécution en appuyant sur les touches Ctrl-F5 produira l'affichage *Bonjour à tous*.

2. Les variables simples.

2.1. Introduction.

Pour des questions de sécurité, le langage C# est un langage très typé. Les variables sont déclarées comme étant d'un type particulier et chaque variable est contrainte à contenir uniquement une valeur du type déclaré.

Les variables peuvent contenir soit des types valeur ou des types référence ou éventuellement des pointeurs.

Nous retrouvons au niveau du langage intermédiaire IL des types prédéfinis permettant ainsi de résoudre au niveau du .NET les problèmes liés à l'existence des différents langage de base C# ou VB.NET. Ces types prédéfinis que l'on appelle CTS (Common Type System) se retrouvent au niveau du .NET Framework sous forme de structure.

Prenons la déclaration d'un entier codé sur 32 bits:

En C#, nous retrouvons la syntaxe `int a`

En VB.NET, nous retrouvons la syntaxe `Integer a`

Ces deux syntaxes d'un point de vue CTS se ramènent à la même déclaration équivalente qui serait: `System.Int32 a`. La variable `a` est en fait ici considérée comme une instance de la structure `Int32` appartenant à l'espace de nom `System`. De ce fait, on peut facilement imaginer que cette structure offre des méthodes qui peuvent être appelées par l'instance. Nous pouvons en effet retrouver `a.ToString()` permettant d'assurer la conversion de l'entier vers une chaîne de caractères.

2.2. Les types valeur.

Il est possible en C# de définir nos propres types valeur en les déclarant comme énumérations ou comme structures. La table suivante donne des informations sur les types valeur prédéfinis.

1. Les types entiers.

Nom	CTS	Description	Valeurs possibles
sbyte	System.SByte	Entier signé sur 8 bits	-128:127
short	System.Int16	Entier signé sur 16 bits	-32768:32767

int	System.Int32	Entier signé sur 32 bits	$-2^{31} : 2^{31} - 1$
long	System.Int64	Entier signé codé sur 64bits	$-2^{63} : 2^{63} - 1$
byte	System.Byte	Entier non signé codé sur 8 bits	0:255
ushort	System.UInt16	Entier non signé sur 16 bits	0:65535
uint	System.UInt32	Entier non signé sur 32 bits	$0 : 2^{32} - 1$
ulong	System.UInt64	Entier non signé codé sur 64 bits	$0 : 2^{64} - 1$

2. Les types flottants.

Nom	CTS	Description	Valeurs possibles
Float	System.Single	Nombre en virgule flottante sur 32 bits en simple précision	
Double	System.Double	Nombre en virgule flottante sur 64 bits en double précision	

3. Le type décimal.

Nom	CTS	Description	Valeurs possibles
decimal	System.Decimal	Nombre en virgule flottante sur 128 bits en haute précision	

4. Le type booléen.

Nom	CTS	Description	Valeurs possibles
Bool	System.Boolean		true ou false

Contrairement au langage C ou un booléen n'existe pas comme type mais provient de la création d'un type énuméré, on retrouve en C# un type booléen natif. En C#, il est impossible de convertir implicitement les valeurs bool en entiers.

5. Le type caractère (char).

Nom	CTS	Description	Valeurs possibles
Char	System.Char	représente un seul caractère codé sur 16 bits(Unicode)	

Du fait qu'un caractère soit codé sur 16 bits, il n'est pas possible d'effectuer une conversion implicite vers un type byte.

2.3. La déclaration des variables de type valeur et leur initialisation.

Nous pouvons déclarer une variable de type valeur à l'aide de la syntaxe suivante:

datatype *identifiant*;

Exemples:

```
int a;
bool y;
```

On peut initialiser une variable en utilisant l'opérateur d'affectation '='. Cette initialisation peut s'effectuer en même temps que la déclaration en utilisant la forme suivante:

```
int a = 10;
bool y = true;
```

Par défaut, une constante entière est considérée comme le compilateur comme étant de type entier. Pour forcer le compilateur à considérer les constantes d'un autre type, il faudra lors de l'initialisation utiliser les syntaxes suivantes:

```
uint x = 123U; //U comme unsigned
long y = 1235L; //L comme long
ulong z = 256UL //UL unsigned long
```

Par défaut, une constante réelle est considérée en double précision. Pour forcer le compilateur à considérer les constantes d'un autre type, il faudra lors de l'initialisation utiliser les syntaxes suivantes:

```
decimal a = 13.20M;
float b = 15.75F; //F comme float
```

Par défaut, certaines variables sont initialisées et d'autres ne le sont pas. Le compilateur C# met l'accent sur la sécurité et exige que toutes les variables soient initialisées avec une valeur de départ avant que le programmeur puisse s'y référer dans une opération. Les violations à ces règles sont traitées comme des erreurs lors de la compilation.

Voici les règles d'initialisation par défaut:

- ❑ Les variables correspondant à des membres dans une classe ou une structure sont remises à zéro par défaut au moment de leur création si elles ne sont pas explicitement initialisées.
- ❑ Les variables qui sont locales dans une méthode doivent être explicitement initialisées.

Reprenons l'exemple suivant:

```
1. using System;
2. namespace MyNameSpace
3. {
4.     public class MyFirstApp
5.     {
6.         public static void Main()
7.         {
8.             int x;
9.             Console.WriteLine(x);
10.        }
11.    }
12. }
```

La ligne 9 présente une erreur à la compilation car la variable x est locale à la méthode Main() et n'est pas initialisée par défaut. Pour éviter cette erreur, il suffit de remplacer la ligne 8 par `int x = 0;`

2.4. La portée des variables.

Ce que l'on entend par portée, c'est la partie de code dans laquelle une variable est accessible. On peut déterminer la portée selon les règles suivantes:

- ❑ Un membre d'une classe est visible aussi longtemps que la classe qui le contient est dans la portée.
- ❑ Une variable locale est visible uniquement dans la partie de code où elle a été déclarée, ce qui correspond à la fermeture de l'accolade du bloc où elle a été déclarée. Peuvent être considérées comme variables locales celles qui sont déclarées dans une instruction `for` et `while`.

2.5. Les types référence.

1. Présentation.

Les types référence prédéfinis sont *objet* et *string* où *objet* est la classe de base de tous les autres types. Des nouveaux types référence peuvent être définis en utilisant les mots clefs *class*, *interface* ou *delegate* qui seront étudiés dans un paragraphe ultérieur.

Le type référence stocke l'adresse de la zone mémoire occupée par l'objet qu'il référence. L'objet occupe une zone que l'on appelle le tas managé tandis que l'adresse est placée dans la pile au même titre qu'un type valeur.

Exemple:

```

1.  using System;
2.
3.  namespace MyNameSpace
4.  {
5.      class MyFirstObjet
6.      {
7.          public int a;
8.          public void print()
9.          {
10.             Console.WriteLine("Contenu du membre a:"+a.ToString());
11.          }
12.      }
13.      class MyFirstApp
14.      {
15.          static void Main()
16.          {
17.             MyFirstObjet x = new MyFirstObjet();
18.             x.a = 10;
19.             x.print();
20.          }
21.      }
22.  }
```

Nous retrouvons dans notre exemple une classe que l'on va instancier grâce à l'opérateur *new*. Cet opérateur permet la création de l'instance et retourne une adresse qui sera placée dans la variable *x* de type référence. On retrouve l'opérateur *new* mais nul part dans notre code l'opérateur *delete* qui permettrait de libérer l'espace alloué à l'objet.

Ce n'est pas une erreur de programmation car, dès qu'un objet présent sur le tas managé n'est plus instancié, c'est le ramasse-miettes (garbage collector) qui s'occupe de libérer automatiquement cet espace.

Nous pouvons étendre notre exemple aux aspects suivants:

```

13.      class MyFirstApp
14.      {
```

```

15.         static void Main()
16.         {
17.             MyFirstObjet x = new MyFirstObjet();
18.             MyFirstObjet y = x;
19.             x.a = 10;
20.             y.print();
21.         }
22.     }

```

L'exécution du code modifié dans notre code modifié de la sorte provoque le même affichage prouvant que x et y pointent vers le même objet.

2.6. Le type prédéfini string.

Les programmeurs en C ont certainement eu recours aux fonctions de gestion de chaînes de caractères dont les prototypes se trouvaient définis dans le fichier d'en-tête string.h.

En C++, la manipulation des chaînes se simplifie en ayant recours aux instanciations de la classe Cstring. Exemples :

En langage C :

```

char first[50] ;
char second[50] ;
strcpy(first, "bonjour");
strcpy(second, " à tous");
strcat(first,second);

```

En Visual C++

```

Cstring first, second;

first = "bonjour";
second= " à tous";
first = first + second;

```

En C#, voici un exemple complet:

```

using System;

namespace MyFirstApp
{
    class MyFirstClass
    {
        static void Main()
        {
            string first = "bonjour";
            string second = " à tous";
            string result = first + second;
            Console.WriteLine("resultat:" + result);
        }
    }
}

```

L'exécution du programme donne alors l'affichage: resultat:bonjour à tous. Comme string est un type prédéfini, il n'est pas nécessaire dans ce cas d'utiliser l'opérateur new.

Sachant que string est un type référence, qu'en est t il de l'exemple suivant:

```

1.  using System;
2.
3.  namespace MyFirstApp
4.  {
5.      class MyFirstClass
6.      {
7.          static void Main()
8.          {

```

```

9.         string first = "chaîne initiale";
10.        string second = first;
11.        second = "chaîne modifiée";
12.        Console.WriteLine("resultat:" + first);
13.    }
14.    }
15. }

```

La ligne 9 nous permet de créer un type référencé qui contiendra l'adresse de l'instance de l'objet String placé sur le tas mangé. A la ligne suivante, nous créons un deuxième type référence que nous initialisons avec la même adresse. On peut donc supposer que les deux types pointent vers le même objet et que la modification de l'un induira automatiquement une modification sur l'autre.

Bien que le type string soit un type référence, sa fonctionnalité est toute différente. Lorsque l'objet est initialisé, il a une taille permettant de contenir juste la chaîne de départ, à savoir *chaîne initiale*. Si on veut remplacer cette chaîne par une autre, une nouvelle instance est créée et la nouvelle adresse se retrouve alors dans la variable *second*.

L'affichage résultant de l'exécution du programme donnera donc:
resultat:chaîne initiale.

3. Les tableaux.

3.1. Introduction.

L'approche que l'on fait des tableaux en C et en C++ est tout à fait différente de celle du C# : en C#, un tableau est une instance de la classe de base .NET System.Array. Cette instantiation permet de mieux contrôler les accès et d'éviter les erreurs fréquentes de débordement des limites du tableau. Ces erreurs apparaissaient lors de l'exécution du programme et provoquaient souvent un plantage de ce dernier.

3.2. Les tableaux à 1 dimension.

Pour déclarer un tableau en C#, il suffit de fixer un jeu de crochets droits à la fin du type de variable des éléments individuels. Notons que tous les éléments d'un tableau sont du même type de données.

Exemple:

```

1.  using System;
2.
3.  namespace MyFirstApp
4.  {
5.      class MyFirstClass
6.      {
7.          static void Main()
8.          {
9.              int[] tabint = new int[10];
10.             float[] tabfloat = null;
11.             tabint[0] = 10;
12.             tabfloat = new float[10];
13.             tabfloat[9] = 4.52F;
14.             Console.WriteLine("entier[0]:" + tabint[0].ToString());

```

```

15.             Console.WriteLine("flottant[9]:"+tabfloat[9].ToString());
16.         }
17.     }
18. }

```

Nous retrouvons en ligne 9 la déclaration d'un tableau d'entier. La variable `tabint` est de type référence : nous utilisons en effet l'opérateur `new` pour instancier la classe. Dans notre exemple, nous avons déclaré un tableau d'entier de 10 cases. Tout comme pour le C et le C++, l'indice le plus petit avec lequel on peut accéder au tableau est 0. Nous pourrions donc aller de l'indice 0 à l'indice 9.

A la ligne 10, nous avons déclaré uniquement le type référence et nous l'avons initialisé avec l'adresse `null` tandis que l'instanciation s'effectue à la ligne 12.

Si vous essayez d'accéder à un indice dépassant les possibilités de taille du tableau, une exception sera générée à l'exécution du programme.

Vous pouvez également déclarer un tableau de chaînes de caractères en utilisant la syntaxe suivante:

```

1. using System;
2.
3. namespace MyFirstApp
4. {
5.     class MyFirstClass
6.     {
7.         static void Main()
8.         {
9.             string[] tabchaine = new string[10];
10.            string[] tabsec = {"un", "deux", "trois"};
11.            Console.WriteLine(tabsec[2]);
12.        }
13.    }
14. }

```

La ligne 9 permet de déclarer un tableau de 10 chaînes de caractères. La ligne 10 permet de déclarer et d'initialiser en une seule ligne d'instruction le tableau. L'exécution du programme provoquera l'affichage suivant: trois.

Il n'est pas possible de modifier la taille d'un tableau une fois qu'il a été instancié.

Etant donné qu'un tableau correspond à un objet que l'on doit instancier, on pourra donc avoir accès à une série de méthodes statiques permettant de gérer ce tableau de chaînes de caractères. Si vous désirez trier le tableau, vous pourrez utiliser la méthode `Sort()`. Attention car une méthode statique ne peut être appelée que par le nom de la classe.

Exemple de modification du code ci dessus:

```

Array.Sort(tabsec);
Console.WriteLine(tabsec[2]);

```

Le résultat de l'exécution du programme donnera donc: un

Voici un autre programme permettant le passage en revue du tableau et l'affichage des différentes chaînes.

```

1. using System;
2.
3. namespace MyFirstApp
4. {
5.     class MyFirstClass

```

```

6.         {
7.             static void Main()
8.             {
9.                 string[] tabchaine = new string[10];
10.                string[] tabsec = {"un", "deux", "trois"};
11.                Array.Sort(tabsec);
12.                foreach(string compte in tabsec)
13.                    Console.WriteLine(compte);
14.            }
15.        }
16.    }

```

L'exécution du programme donnera l'affichage suivant:

```

deux
trois
un

```

Les autres tableaux n'échappent pas à cette façon de faire.

```

1.     class MyFirstClass
2.     {
3.         static void Main()
4.         {
5.             int[] i = new int[] {1,2};
6.             int[] j = {1,2};
7.             Console.WriteLine(i[0]);
8.             Console.WriteLine(j[0]);
9.         }
10.    }
11. }

```

3.3. Les tableaux multidimensionnels.

C# supporte deux types de tableaux multidimensionnels: un tableau rectangulaire appelé aussi matrice et un tableau orthogonal (appelé jagged). Un tableau rectangulaire a un nombre constant de colonne pour l'ensemble des lignes tandis qu'un tableau orthogonal peut avoir un nombre variable de colonne pour chacune des lignes qui le compose.

a) les tableaux rectangulaires.

La syntaxe suivante permet de déclarer un tableau rectangulaire de 2 lignes et 4 colonnes :
`int[,] tabrect = new int[2,3];` // la virgule indique que le tableau a deux dimensions.

Tout comme pour les tableaux unidimensionnels, la déclaration peut se faire lors de l'initialisation en utilisant la syntaxe suivante :

```

int[,] tabrect = {{1, 2, 3}, {4, 5, 6}};
string[,] tabrect = {"Dupond","Albert"}, {"Dubart","Eric"}, {"Durant","Thierry"}};

```

Le tableau précédent possède 2 lignes et 3 colonnes.

Etant donné que le tableau multidimensionnel correspond à une instance de la classe `System.Array`, nous disposons de méthodes permettant de récupérer des informations sur le tableau ou de pouvoir demander d'en trier le contenu.

Exemple :

```
1.     class MyFirstClass
2.     {
3.         static void Main()
4.         {
5.             int[,] tabint = new int[2,3];
6.             int [,] tabint2 = {{4,7,5},{8,2,12}};
7.             for (int i=0; i<=tabint2.GetUpperBound(0);i++)
8.                 for (int j=0;
9.                     j<=tabint2.GetUpperBound(1);j++)
10.                    Console.WriteLine(tabint2[i,j]);
11.         }
```

La ligne 7 et 8 reprennent l'appel à la méthode `GetUpperBound` : la méthode renvoie l'indice maximal de la dimension envoyée en paramètre. Ceci nous évite de devoir mémoriser les indices maximaux autorisés pour l'ensemble des tableaux avec lesquels on nous travaillons.

b) Les tableaux orthogonaux.

La syntaxe suivante nous permet de déclarer un tableau orthogonal :

```
int[][] tabjag = new int[2][];
tabjag[0] = new int [4];
tabjag[1] = new int [6];
```

Attention à la différence de syntaxe:

`int[,] x = new int [4,7]` ; permet la création d'un tableau rectangulaire

`int[][] y = new int[2][]` ; permet la création d'un tableau orthogonal comprenant deux lignes mais le nombre de colonne est variable. Ce type de tableau peut être vu comme un tableau de tableau.

Une autre syntaxe permettant l'initialisation en même temps que la déclaration nous donnerait donc :

```
int[][] tabjag = new int[][] {new int[] {1, 2, 3, 4}, new int[] {5, 6, 7, 8, 9, 10}};
```

Tout comme les tableaux rectangulaires, l'on pourra utiliser les différentes méthodes mises à notre disposition.

```
1.     class MyFirstClass
2.     {
3.         static void Main()
4.         {
5.             int[][] tabjag = new int[][] { new int[] {1, 2,
6.                 3, 4}, new int[] {5, 6, 7, 8, 9, 10}};
7.             for (int i=0; i<tabjag.GetLength(0); i++)
8.                 for (int j=0;
9.                     j<tabjag[i].GetLength(0);j++)
10.                    Console.WriteLine(tabjag[i][j]);
11.         }
```

A la ligne 6 et 7, nous retrouvons l'appel à la méthode `GetLength` qui renvoie le nombre d'éléments que l'on retrouve dans une dimension passée en paramètre.

3.4. Les conversions de type.

Bien que fort semblable à ce que l'on connaît en langage C, il nous a semblé important de redonner des notions simples sur les conversions de type. Vous devez retenir que lorsque l'on essaie de copier le contenu d'une variable dans une autre par un opérateur d'affectation, si vous ne risquez pas de perdre d'informations utiles, la conversion est dite implicite et le compilateur ne génère pas d'erreurs.

Si l'opération d'affectation risque de vous faire perdre des informations utiles, le compilateur nécessite l'utilisation d'un opérateur de transtypage.

Exemple:

```
long a = 10;
int b = a;
```

Bien que la valeur 10 puisse tenir dans la variable b, le compilateur n'accepte pas que vous placiez dans un entier codé sur 32 bits, le contenu d'un entier codé sur 64 bits. Pour que cela puisse être compilé sans erreur, vous devez utiliser l'opérateur de transtypage sous la syntaxe suivante:

```
long a = 10;
int b = (int) a;
```

Si vous souhaitez qu'un transtypage forcé ne provoque pas de débordement qui pourrait rendre l'exécution de votre programme imprévisible, il est possible de forcer le runtime à lever une exception de dépassement en cas de besoin.

Exemple:

```
int a = 525;
byte b = (byte) a;
Console.WriteLine("contenu de b: "+b.ToString( ));
```

Le compilateur ne génère pas d'erreur et pourtant un tel code exécuté donne un résultat prévisible mais qui peut ne pas vous satisfaire. Vous voyez apparaître sur votre écran *contenu de b: 13*. Vous obtenez en fait 525 modulo 256 car un byte est codé sur 8 bits.

Si vous souhaitez que le runtime génère une exception, il faudra utiliser la syntaxe suivante:

```
int a = 525;
byte b = checked ((byte) a);
Console.WriteLine("contenu de b: "+b.ToString( ));
```

Voici un tableau des conversions qui sont implicites:

De	Vers
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double,

decimal

Pour les conversions entre un nombre et une chaîne, nous disposons des méthodes que nous avons eu l'occasion d'utiliser à maintes reprises. La classe de base `Objet` implémente en effet une méthode appelée `ToString()`.

Pour parser une chaîne afin de retrouver une valeur numérique ou booléenne, nous pouvons utiliser la méthode `Parse()` prise en compte par tous les types valeur prédéfinis:

```
string s = "100,20";
float x = float.Parse(s);
Console.WriteLine(x);
```

4. Les énumérations

Une énumération est un type entier défini par l'utilisateur. La syntaxe est identique à celle que l'on retrouve dans le langage C ou C++. Lorsque nous définissons une énumération, nous fournissons du texte qui est alors utilisé comme constante pour leurs valeurs correspondantes. Voici un exemple:

```
1.  class MyFirstClass
2.  {
3.      public enum DayOfWeek
4.      {
5.          Dimanche = 1,
6.          Lundi,
7.          Mardi,
8.          Mercredi,
9.          Jeudi,
10.         Vendredi,
11.         Samedi
12.     }
13.     static void Main()
14.     {
15.         DayOfWeek Day;
16.         Day = DayOfWeek.Dimanche;
17.         Array dayArray = Enum.GetValues(typeof(DayOfWeek));
18.         foreach (DayOfWeek Days in dayArray)
19.         {
20.             Console.WriteLine("Le jour n°{0} est le
21.                                 {1}", (int)Days, Days);
22.         }
23.     }
```

Un type énuméré dépend de `System.Enum` et de ce fait, toute variable de type énuméré doit être considérée comme un objet. Cette classe possède des méthodes statiques que l'on utilise dans notre code pour notamment récupérer un tableau des valeurs et des constantes d'un type énuméré particulier. Il s'agit de `Enum.GetValues` que l'on retrouve à la ligne 17.

`typeof` est un opérateur qui retourne un objet `Type` représentant un type spécifié. Dans notre cas, `typeof(DayOfWeek)` renvoie un objet `Type` représentant le type `System.Enum`.

Nous retrouvons à la ligne 18 l'instruction `foreach` qui permet de passer en revue un tableau d'objet, et de les retourner un par un au moyen de l'objet `Days`.

Nous retrouvons à la ligne 20 une syntaxe différente de la méthode `WriteLine` qui va nous permettre d'envisager l'utilisation de plusieurs arguments.

```
Console.WriteLine("Le jour n°{0} est le {1}", (int)Days, Days);
```

{0} correspond au premier argument dans la liste c-à-d (int)days.
{1} correspond au deuxième argument dans la liste c-à-d Days.

5. C# orienté objet.

5.1. Syntaxe des classes.

La syntaxe des classes en C# est fort similaire à celle du C++. Nous y retrouvons des déclarations de variables membres et des déclarations de méthodes encore appelées fonctions membre. Il est important malgré tout de signaler que l'utilisation du mot clef *struct* lors de la déclaration de la classe a d'autres effets que ceux présentés en C++, à savoir de rendre par défaut les membres et les méthodes publics. Nous y reviendrons lors d'un paragraphe ultérieur et nous n'utiliserons dans un premier temps que le mot clef *class*.

Les modificateurs globaux de type public, private ou protected n'existent plus dans le sens où nous devons maintenant faire précéder chaque membre ou méthode de ce mot clef pour l'en affecter.

```
Class MyFirstClass
{
    private int UnMembre ;
    public void UneMethode(bool UnParamètre)
    {
    }
}
```

En C++ les méthodes pouvaient être définies à l'extérieur de la classe en utilisant l'opérateur de résolution de portée ; en C# la définition des méthodes doit se réaliser à l'intérieur de la classe. La création d'un objet en C# s'effectue toujours par référence c-à-d que vous devez nécessairement utiliser l'opérateur new sous la forme suivante :

```
MyFirstClass MonObject = new MyFirstClass();
```

Pour rappel, d'un point de vue occupation mémoire, la référence est placée sur la pile tandis que l'objet référencé est placé sur le tas managé.

5.2. L'héritage simple.

La notion d'héritage a été fortement simplifiée par rapport à ce que l'on connaît du C++ : il n'y a plus que de l'héritage simple, l'héritage multiple ayant été abandonné. Voici la syntaxe à utiliser pour mettre en place l'héritage simple :

```
Class MyDerivedclass : MyBaseClass
{
    //membres et fonctions à placer ici.
}
```

Nous remarquerons à ce niveau-ci une différence avec le C++ car l'héritage ne comprend plus de modificateur d'accessibilité.

5.3. La déclaration des méthodes.

En C#, la déclaration des méthodes comprend le modificateur d'accès. Nous retrouvons la syntaxe suivante lors de la déclaration :

```
[modificateur] type_de_retour Identificateur([paramètres])
{
    // corps de la fonction.
}
```

Nous retrouverons les modificateurs suivants :

Static. Il existe deux grandes catégories de méthodes : les méthodes dites instanciées et les méthodes dites de classe. Les méthodes instanciées n'existent que si l'on a créé une instance de la classe et elles sont appelées par l'objet lui-même sous la forme : *objet.méthode([paramètres])* ;. Les méthodes de classe déclarées avec le mot clef `static` peuvent être appelées même s'il n'existe aucune instance de la classe sous la syntaxe suivante : *Nom_de_classe.Méthode ([paramètres])* ;.

Prenons comme exemple une classe permettant la gestion des nombres complexes. Sans mettre en œuvre la surcharge des opérateurs, nous allons intégrer dans notre classe une méthode permettant d'effectuer une opération d'addition sur deux complexes. En C++, nous pourrions utiliser une fonction amie de sorte que les objets passés en paramètre puissent accéder aux membres protégés de la classe mais en C#, cette solution n'est pas autorisée. Nous devons donc nous orienter vers des méthodes statiques. Un peu dans le même esprit que pour les fonctions amies, l'ensemble des objets doit être passé en paramètre du fait que la méthode statique est appelée par le nom de la classe et pas par une instance.

```
1 namespace ConsoleApp
2 {
3     class Complexe
4     {
5         public int reel;
6         public int image;
7         public Complexe(int reel, int image)
8         {
9             this.reel=reel;
10            this.image=image;
11        }
12        public static Complexe MultComplexe(Complexe x, Complexe y)
13        {
14            int reel=x.reel*y.reel-x.image*y.image;
15            int image=x.reel*y.image+x.image*y.reel;
16            Complexe tmp = new Complexe (reel,image);
17            return tmp;
18        }
19    }
20    class Class1
21    {
22        static void Main(string[] args)
23        {
24            Complexe ca = new Complexe(10,20);
25            Complexe cb = new Complexe(20,30);
26            Complexe cc = Complexe.MultComplexe(ca,cb);
27            Console.WriteLine("{0}+i{1}",ca.reel.ToString(),
28                               ca.image.ToString());
29        }
30    }
```

Remarque importante de syntaxe:

En C++, l'opérateur `this` renvoyait dans une méthode un pointeur sur l'instance l'ayant appelée. En C#, nous ne parlons plus de pointeurs mais de référence et de ce fait, `this` n'échappe pas à la règle. D'un point de vue syntaxe, on retrouve la différence suivante aux lignes 9 et 10 :

En C++	En C#
<code>this->reel</code>	<code>this.reel</code>
<code>this->image</code>	<code>this.image</code>

new, virtual, override. Ces modificateurs concernent l'héritage. Lorsque le mot clef **virtual** est utilisé pour déclarer une méthode d'une classe de base, cela signifie qu'elle est surchargée dans les classes dérivées. Dans les classes dérivées, la surcharge de la méthode de la classe de base doit être déclarée en utilisant le mot clef **override**. Notons que pour qu'une méthode puisse surcharger une autre méthode, la méthode surchargée ne doit pas être de type `static` et qu'elle doit être déclarée en utilisant le mot clef `virtual`, `abstract` ou `override`.

L'opérateur **new** permet à une méthode d'une classe dérivée de cacher une méthode d'une classe de base sans pour autant nécessiter une déclaration particulière pour la méthode cachée.

Supposons une classe de base appelée `rectangle` et une classe dérivée appelée `carré`. Supposons dans la classe de base une méthode appelée `surface` qui doit être surchargée dans la classe dérivée.

- Utilisation des mots clefs `virtual` et `override`:

```
1 namespace ConsoleApp
2 {
3
4     class Rectangle
5     {
6         protected float longueur;
7         protected float largeur;
8         public Rectangle(float largeur, float longueur)
9         {
10             this.largeur = largeur;
11             this.longueur = longueur;
12         }
13         public float surface()
14         {
15             return largeur*longueur;
16         }
17     }
18     class Carre:Rectangle
19     {
20         public Carre(float largeur):base(largeur,0)
21         {
22         }
23         public float surface()
24         {
25             return largeur*largeur;
26         }
27     }
28 }
29
30 class Class1
31 {
32     static void Main(string[] args)
33     {
34         Carre ca = new Carre(10.0F);
35         Rectangle ra = ca;
```

```

36         Console.WriteLine("surface:"+ra.surface().ToString());
37
38         Console.WriteLine("surface:"+ca.surface().ToString());
39     }
40 }

```

Commentaires:

Pour rappel, en C++, un pointeur d'une classe de base peut contenir l'adresse d'une classe dérivée. En C#, une référence d'une classe de base peut pointer vers un objet d'une classe dérivée. C'est ce que l'on retrouve à la ligne 35. La question à se poser est de savoir quelle est la méthode qui sera appelée à la ligne 36: est ce la méthode liée au type de la référence ou au type de l'objet sur lequel la référence pointe.

L'exécution du code nous donne l'affichage suivant:

```

surface:0
surface:100
Press any key to continue

```

C'est bien le type de la référence et non pas le type de l'objet référencé. Si nous voulons que le choix de la méthode se fasse sur base du type de l'objet pointé, nous devons utiliser le mot clef *virtual* pour définir la méthode de la classe de base et obtenir la syntaxe suivante:

```
virtual public float surface()
```

L'analyse de l'affichage résultant de la compilation nous donne l'avertissement suivant:

```
warning CS0108: The keyword new is required on 'ConsoleApp.Carre.surface()' because it
hides inherited member 'ConsoleApp.Rectangle.surface()'
```

Actuellement, ce message n'a pas beaucoup de signification car nous n'avons pas encore dévoilé les particularités de l'opérateur *new* dans la déclaration d'une méthode.

Si nous nous limitons à l'usage de l'opérateur *virtual*, nous obtenons à la compilation le message suivant:

```
warning CS0114: 'ConsoleApp.Carre.surface()' hides inherited member
'ConsoleApp.Rectangle.surface()'. To make the current member override that implementation,
add the override keyword. Otherwise add the new keyword.
```

L'exécution du code donne le même résultat. Il faut donc ajouter le mot clef *override* dans la déclaration de la surcharge sous la forme:

```
override public float surface()
```

L'exécution de notre programme nous permet maintenant d'obtenir l'affichage suivant:

```

surface:100
surface:100
Press any key to continue

```

- **Utilisation du mot clef *new***

Dans le premier exemple de notre classe Complexe, la méthode surchargée cache par défaut la méthode de la classe de base. L'utilisation du mot clef *new* a le même effet excepté que le compilateur ne génère plus de message d'attention.

Abstract Une méthode abstract n'a pas d'implémentation et doit être surchargée dans toute classe dérivée non abstraite. Evidemment, une méthode abstraite est automatiquement virtuelle. Une *interface* est une classe qui ne contient que des méthodes abstraites. Cependant, de telles classes sont déclarées avec le mot clef *interface* plutôt que le mot clef *class*. Une classe peut hériter d'une seule classe de base mais peut hériter de plusieurs interface. La notion d'interface sera abordée dans un paragraphe ultérieur.

Extern Les méthodes qui sont déclarées comme étant externes sont définies à l'extérieur en utilisant un langage autre que C#.

5.4. Les passages de paramètres.

Dans des langages tels que C ou C++, le passage de paramètres à une fonction ou à une méthode peut se faire par valeur, par pointeur et uniquement en C++ par référence avec l'utilisation de l'opérateur &. En C#, l'utilisation des pointeurs n'étant pas possible, on retrouvera des passages par valeur ou par référence.

- **Passage par référence.**

L'exemple suivant reprend un passage par référence bien que l'entier soit de type valeur.

```
1      class Class1
2      {
3          /// <summary>
4          /// The main entry point for the application.
5          /// </summary>
6          [STAThread]
7          static void Main(string[] args)
8          {
9              int b=10;
10             Console.WriteLine("Avant appel:"+b.ToString());
11             change(ref b);
12             Console.WriteLine("Après appel:"+b.ToString());
13         }
14
15         static void change (ref int a)
16         {
17             a=20;
18         }
19     }
```

Remarque concernant la syntaxe:

La ligne 15 reprend le mot clef *ref* dans la déclaration du paramètre et la ligne 11 doit comprendre obligatoirement le mot clef *ref* lors de l'appel.

L'exécution du code provoque l'affichage suivant:

```
Avant appel:10
Après appel:20
Press any key to continue
```

Nous pouvons en déduire que la variable locale `a` est en fait une référence pointant sur le même objet que la variable `b`. Toute modification du contenu de `a` provoque une modification du contenu de `b`.

- **Passage par valeur.**

Idem que dans le langage C et C++.

- **Le modificateur de paramètre "out".**

Lorsque nous désirons qu'une fonction renvoie plusieurs valeurs au programme appelant, nous utilisons les passages par pointeurs si nous travaillons en C++; en C#, nous utiliserons les références. Un des points importants du C# est qu'il faut initialiser les variables même si celles-ci n'ont pour but que de récupérer des valeurs en provenance d'une fonction appelée.

Reprenons l'exemple de la classe `rectangle` et modifions notre code de sorte que la fonction `surface` renvoie la surface du rectangle par paramètre.

```
1     class rectangle
2     {
3         float largeur;
4         float hauteur;
5         public rectangle(float largeur, float hauteur)
6         {
7             this.largeur=largeur;
8             this.hauteur=hauteur;
9         }
10        public void surface(ref float surf)
11        {
12            surf = largeur*hauteur;
13        }
14    }

15    class Class1
16    {
17        static void Main(string[] args)
18        {
19            float result;
20            rectangle ra = new rectangle(10.0F, 2.5F);
21            ra.surface(ref result);
22            Console.WriteLine("surface:"+result.ToString());
23        }
24    }
25 }
```

Dans notre exemple, l'initialisation de la variable locale `result` ne devrait pas être nécessaire puisqu'elle ne sert qu'à recevoir le résultat de l'exécution de la méthode `surface`.

Malgré cette certitude, le compilateur respecte la règle de réclamer l'initialisation de toute variable locale avant son utilisation et nous obtiendrons l'erreur suivante à la compilation:

```
Class1.cs(34): Use of unassigned local variable 'result'
```

Pour éviter ce problème, il suffit d'utiliser le mot clef `out`. Ce mot clef doit être utilisé lors de la déclaration de la méthode et lors de l'appel en remplacement du mot clef `ref`. Il a le même effet de produire un passage par référence mais en tolérant la variable non initialisée lors de l'appel.

surface:25

Press any key to continue

- **Le modificateur de paramètre "params".**

Il arrive que nous désirions faire passer un nombre variable de paramètres lors d'un appel à une fonction; ce passage peut s'effectuer en utilisant le modificateur *params*. Tous les paramètres passés lors de l'appel sont passés par valeur.

```
1      class Class1
2      {
3          static void Main(string[] args)
4          {
5              int a=10,b=20,c=30;
6              float result;
7              result=find(a,b,c);
8              Console.WriteLine("Le plus
9                  grand:"+result.ToString());
9          }
10         static int find(params int[] args)
11         {
12             int greater=args[0];
13             for (int i=1;i<args.Length;i++)
14             {
15                 if (args[i]>greater) greater=args[i];
16             }
17             return greater;
18         }
19     }
20 }
```

Remarque concernant la syntaxe:

La ligne 13 comprend un appel à la propriété *Length* qui renvoie le nombre de paramètres qui ont été passés lors de l'appel de la méthode.

5.5. La surcharge des méthodes.

Tout comme pour le C++, le C# supporte la surcharge des méthodes c-à-d que plusieurs versions d'une même méthode qui ont différentes signatures (nom, nombre des paramètres, types des paramètres) peuvent exister dans la même classe; cette surcharge est également valable pour des méthodes tels que les constructeurs. Le C# n'acceptant pas les paramètres par défaut, la surcharge permet de contourner cette limitation

5.6. Les propriétés.

En C#, les propriétés représentent un concept repris du Visual Basic. Une propriété consiste en une méthode ou paire de méthodes permettant la modification ou la lecture du contenu d'un membre sans y accéder directement.

Pour définir une propriété en C#, nous utiliserons la syntaxe suivante:

```
1      class carre
2      {
3          private float cote;
4          public float Cote
```

```

5         {
6             get
7             {
8                 return cote;
9             }
10            set
11            {
12                cote=value;
13            }
14        }
15        public carre(float cote)
16        {
17            this.cote=cote;
18        }
19    }

20    class Class1
21    {
22        static void Main(string[] args)
23        {
24            carre ca = new carre(12.5F);
25            Console.WriteLine("Lecture:"+ca.Cote);
26            ca.Cote=10.0F;
27            Console.WriteLine("valeur du cote:"+ca.Cote);
28        }
29
30
31    }

```

Si vous désirez qu'une propriété soit accessible uniquement en lecture, il suffit de conserver le membre correspondant en privé et de ne prévoir dans votre classe que l'accessor get.

Soit l'exercice suivant: nous imaginons une classe dans laquelle nous désirons à tout instant pouvoir connaître le nombre d'instances qui ont été créées. Pour ce, reprenons notre classe au point 5.2 permettant de gérer les complexes et implémentons ce membre.

```

1    class Complexe
2    {
3        private static int compteur=0;
4        public int reel;
5        public int image;
6        public static int Compteur
7        {
8            get
9            {
10                return compteur;
11            }
12        }
13        public Complexe(int reel, int image)
14        {
15            compteur++;
16            this.reel=reel;
17            this.image=image;
18        }
19        ~Complexe ()
20        {
21            compteur--;
22        }
23    }

```

```

24     class Class1
25     {
26         static void Main(string[] args)
27         {
28             Complexe ca = new Complexe(10,20);
29             Complexe cb = new Complexe(20,30);
30             Console.WriteLine("Instances:"+Complexe.Compteur);
31         }
32     }

```

Remarques:

A la ligne 3, nous retrouvons la création d'une variable membre de type statique qui est en mode d'accès privé. Ce membre est en fait notre compteur d'instance qui doit donc être commun à toutes les instances; c'est pourquoi nous choisissons de déclarer ce membre comme statique. L'accès en écriture à ce membre s'effectue uniquement en interne par le constructeur qui l'incrémente et par le destructeur qui le décrémente. Il n'est donc pas utile de rendre accessible cette variable membre en écriture de l'extérieur de la classe. Nous plaçons dans notre classe un accesseur que nous appelons *Compteur*; on ne peut pas lui donner le même nom que celui du membre mais il est intéressant qu'il en soit le plus proche pour une question de facilité dans la lisibilité du code. Le membre *compteur* présente la première lettre en minuscule et l'accesseur *Compteur* la première lettre en majuscule.

Il est à remarquer l'emploi d'un accesseur de type statique qui ne dépend donc pas d'une instance mais sera donc appelé par le nom de la classe (voir ligne 30).

5.7. La surcharge des opérateurs.

Un des intérêts du C++ est de pouvoir surcharger les opérateurs pour une question de facilité dans la gestion des opérations arithmétiques et de comparaisons sur les objets d'une classe.

Dans notre exemple de la classe *Complexe*, nous avons mis en place au point 5.3 une méthode statique permettant de pouvoir effectuer la multiplication de deux nombres complexes. Il est clair que si nous avions pu remplacer la ligne de code `Complexe.MultComplexe(ca,cb)` par `cc=ca*cb;` nous utilisons un opérateur déjà connu qui est l'opérateur de multiplication `*` et le code ne demande que peu de commentaires excepté si nous nous efforçons à choisir des noms de méthodes évocateurs.

La mise en place des surcharges est fort semblable à celle utilisée en C++, excepté que le C# n'accepte pas l'utilisation de fonction amie. Les surcharges doivent donc apparaître comme des méthodes de la classe et les différentes opérands doivent être passées en paramètre. Le nom de la méthode doit comprendre le mot clef *operator* suivi du symbole utilisé pour l'opérateur. Cette méthode particulière doit être déclarée en `static`.

Considérons les deux exercices suivants:

- Remplacer la méthode statique `MultComplexe` par une surcharge de l'opérateur `*`
- Mettre en place la surcharge de l'opérateur `==` permettant de tester l'égalité entre deux nombres complexes.

```

1     class Complexe
2     {
3         public int reel;
4         public int image;
5         public Complexe(int reel, int image)
6         {
7             this.reel=reel;
8             this.image=image;
9         }
10        public static Complexe operator *(Complexe x, Complexe y)
11        {

```

```

12         int reel=x.reel*y.reel-x.image*y.image;
13         int image=x.reel*y.image+x.image*y.reel;
14         Complexe tmp = new Complexe (reel,image);
15         return tmp;
16     }
17     public static bool operator ==(Complexe x, Complexe y)
18     {
19         if ((x.reel==y.reel)&&(x.image==y.image))return true;
20         else return false;
21     }
22     public static bool operator !=(Complexe x, Complexe y)
23     {
24         if ((x.reel==y.reel)&&(x.image==y.image))return false;
25         else return true;
26     }
27
28 }
29 class Class1
30 {
31     static void Main(string[] args)
32     {
33         Complexe ca = new Complexe(10,20);
34         Complexe cb = new Complexe(20,30);
35         Complexe cc = ca * cb;
36
37         Console.WriteLine("resultat:{0}+i{1}",cc.reel,cc.image);
38         if (ca == cb) Console.WriteLine("Complexes égaux");
39         else Console.WriteLine("Complexes différents");
40     }
41 }

```

Remarques:

Les lignes 35 et 37 mettent en évidence l'intérêt d'utiliser les opérateurs dans la simplicité de syntaxe utilisée pour multiplier ou comparer deux objets qui sont ici des instances de ma classe Complexe.

Les lignes 17-21 comprennent la surcharge de l'opérateur ==. En C++ nous aurions pu choisir une méthode retournant un entier car le type natif booléen n'existait pas et de ce fait, les types énumérés BOOL étaient associés à des entiers: l'entier 0 correspondait à une valeur fausse tandis que l'entier 1 ou différent de 0 était associé à une valeur vraie. En C#, ce n'est plus autorisé: une instruction if a besoin d'un type bool en paramètre et pas d'un type entier. Il n'existe pas de conversions implicites entre un entier et un type bool.

Le C# tout comme le C++ demande la surcharge des deux opérateurs == et !=: l'une ne peut être réalisée sans l'autre.

Nous reprenons dans le tableau suivant les opérateurs qui peuvent être surchargés:

Catégorie	Opérateurs	Restrictions
Arithmétique binaire	+,*,/,-,%	aucune
Arithmétique unaire	++,--	aucune
Bit à bit binaire	&,&,^,<<,>>	aucune
Bit à bit unaire	!,~	aucune
Comparaison	==,!=,>,<,<=,>=	surchargée par paire

Certains seront étonnés de ne plus retrouver l'opérateur [] que l'on pouvait surcharger en C++. Cette impossibilité du C# est compensée par l'utilisation possible des indexeurs.

5.8. Les indexeurs.

Un exemple classique de l'utilisation de la surcharge de l'opérateur [] en C++ était la possibilité de pouvoir accéder à un élément d'un vecteur étant lui-même un objet d'une classe.

Reprenons cet exemple et adaptons le à la lumière du C#:

```
1     class vecteur
2     {
3         int [] tab;
4         int taille;
5         public int this [int i]
6         {
7             get
8             {
9                 if (i<taille)
10                return tab[i];
11                else
12                throw new IndexOutOfRangeException("Acces");
13            }
14            set
15            {
16                if (i<taille)
17                tab[i]= value;
18                else
19                throw new IndexOutOfRangeException("Acces");
20            }
21        }
22        public vecteur(int taille)
23        {
24            tab = new int[taille];
25            this.taille=taille;
26        }
27    }
28    class Class1
29    {
30        static void Main(string[] args)
31        {
32            vecteur x = new vecteur(10);
33            x[1]=10;
34            x[9]=12;
35            Console.WriteLine("x[1]="+x[1]);
36        }
37    }
```

Remarques:

L'utilisation d'un indexeur est fort semblable à la mise en place des propriétés accessibles par les accesseurs set et get (get pour la lecture, set pour l'écriture).

Nous retrouvons à la ligne 5 la déclaration de la méthode liée à l'indexeur sous la syntaxe suivante:

public int this [int i] L'opérateur *this* permet de référencer l'instance ayant servie à utiliser l'indexeur. Si nous avons envisagé une classe permettant la gestion d'une matrice avec l'accès à un tableau à deux dimensions, nous aurions eu la syntaxe *public int this [int i, int j]*.

Une particularité aux lignes 12 et 19 est l'emploi d'une syntaxe permettant de générer une exception qui sera prise en charge par le runtime et indiquant qu'il y a eu une tentative d'accès en dehors des limites de notre vecteur. Si dans notre exemple, nous essayons d'entrer la ligne de code suivante: *x[10]=20*; nous obtiendrons lors de l'exécution le message d'erreur suivant:

```
Unhandled Exception: System.IndexOutOfRangeException: Acces
```

5.9. Les constructeurs statiques.

Une nouvelle caractéristique de C# est de permettre l'écriture d'un constructeur statique sans paramètre. Ce constructeur n'est appelé qu'une seule fois mais le runtime ne garantit pas à quel moment ce constructeur est exécuté ni l'ordre dans lesquels ils le seront si plusieurs constructeurs existent dans différentes classes. Nous pouvons simplement dire que le constructeur sera en général exécuté immédiatement avant le premier appel à un des membres de la classe.

Comme c'est le runtime qui appelle ce constructeur, nous ne retrouverons pas de paramètre ni de modificateurs d'accès de type *public* ou *private* qui n'auront pas de sens.

L'utilisation de tels constructeurs peut se comprendre lorsque l'on doit initialiser des membres déclarés en *static*.

Nous retrouverons dans le paragraphe suivant la mise en place d'un tel constructeur.

5.10. Les champs en lecture seule.

Dans la norme ANSI du langage C, il était déjà possible de déclarer une variable en la faisant précéder du mot clef 'const'. Ce mot clef permettait de déclarer une constante dont l'initialisation devait se faire lors de la déclaration.

Dans le langage C#, il est possible de déclarer une variable membre d'une classe avec le mot clef '*readonly*' permettant de rendre ce membre accessible en lecture seule. L'initialisation d'un tel membre peut se faire uniquement dans le constructeur pour les membres instanciés et lors de la déclaration ou dans un constructeur '*static*' pour les membres de classe déclarée avec le mot clef '*static*'.

Si nous reprenons notre exemple permettant la gestion des nombres complexes, nous pourrions ajouter un membre statique renseignant le nombre maximum d'instance que l'on autoriserait à créer.

```
1   class Complexe
2   {
3       public int reel;
4       public int image;
5       public static readonly uint MaxComplexes;
6       public Complexe(int reel, int image)
7       {
8           this.reel=reel;
9           this.image=image;
10      }
11      static Complexe ()
12      {
13          MaxComplexes=20;
14      }
15  }
16  class Class1
17  {
18      static void Main(string[] args)
19      {
20          Complexe x = new Complexe(10,20);
21          Console.WriteLine(Complexe.MaxComplexes);
22          Complexe.MaxComplexes=30;
23      }
24  }
```

Remarques:

A la ligne 5, nous retrouvons un membre de classe déclaré avec le mot clef *static*, le rendant accessible en lecture seule. Nous avons toutefois tenté d'y accéder à la ligne 21 dans la méthode *main* et nous obtenons une erreur à la compilation:

```
...Class1.cs(26): A static readonly field cannot be assigned to (except in a static constructor or a variable initializer)
```

A la ligne 11, nous retrouvons un constructeur assez particulier puisqu'il est précédé du mot clef *static*. C'est ce que nous appelons un constructeur statique dont les caractéristiques essentielles ont été données lors d'un paragraphe précédent.

5.11. Les classes abstraites, les méthodes abstraites et les interfaces

Certaines classes ne sont pas désignées pour être instanciées. Au contraire, elles sont désignées pour être simplement héritées d'une classe dérivée pouvant avoir elle-même des instances. Nous utilisons lors de la déclaration de la classe le mot clef *abstract*.

Dans une classe, nous pouvons également retrouver des méthodes ayant le modificateur *abstract*; celles-ci n'ont pas de définition mais juste une déclaration. Ces classes ne peuvent pas être non plus dérivées puisque de telles méthodes ne peuvent pas être instanciées.

Une 'interface' est une classe qui ne présentera que des méthodes abstraites. Cependant, de telles classes ne sont pas déclarées avec le mot clef *class* mais avec le mot clef *interface*. Une classe dérivée ne peut hériter que d'une seule classe de base mais elle peut hériter de plusieurs interfaces. Une interface définit une sorte de contrat. Une classe ou une structure qui implémente une interface doit adhérer à ce contrat. Les interfaces peuvent contenir des méthodes, des événements et des indexeurs comme membres.

Nous mettrons en place différentes classes permettant d'effectuer des mesures de surface et de périmètre sur des formes géométriques de type parallélépipédiques.

Nous devons retrouver une classe spécifique pour chaque forme mais nous souhaitons mettre en place dans chaque classe un nombre de méthodes identiques permettant leur usage. Une façon simple d'y arriver est d'utiliser une interface.

```
1     interface formes
2     {
3         double surface();
4         double perimetre();
5     }
6
7     class carre:formes
8     {
9         public double cote;
10    }
11
12    class Class1
13    {
14        static void Main(string[] args)
15        {
16        }
17    }
```

Remarques:

Nous retrouvons à la ligne 7 la syntaxe permettant à une classe d'hériter d'une interface. Dès que cet héritage est mis en place, il est indispensable de déclarer une méthode *surface()* et *perimetre()* dans la classe *carre* faute de quoi, le compilateur nous donne les erreurs suivantes:

```
class1.cs(14,8): error CS0535: 'ConsoleApplication8.carre' does not implement interface member 'ConsoleApplication8.formes.surface()'
class1.cs(14,8): error CS0535: 'ConsoleApplication8.carre' does not implement interface member 'ConsoleApplication8.formes.perimetre()'
```

Modifions notre exemple de sorte de faire apparaître les méthodes souhaitées:

```
1     interface Iformes
2     {
3         double surface();
4         double perimetre();
5     }
6
7     class carre:Iformes
8     {
9         double cote;
10        public carre(double cote)
11        {
12            this.cote=cote;
13        }
14        public double surface()
15        {
16            return (cote*cote);
17        }
18        public double perimetre()
19        {
20            return (cote*4);
21        }
22    }
23
24    class Class1
25    {
26        static void Main(string[] args)
27        {
28            carre ca = new carre(10.5);
29            Console.WriteLine(ca.surface());
30            perimetre(ca);
31        }
32    }
33    static void perimetre(Iformes x)
34    {
35        Console.WriteLine(x.perimetre());
36    }
37
38 }
```

Remarques:

Une des caractéristiques des interfaces est que l'on peut créer des références de ces interfaces capables de pointer vers n'importe quelle instance d'une classe héritant de cette instance. La ligne 33 comprend un passage par référence sur l'interface *Iformes*.

On peut faire en sorte que certaines méthodes ne puissent être appelées que par une référence de l'interface. Il suffit pour cela de déclarer la méthode dans la classe en utilisant la syntaxe suivante: la ligne 14 doit par exemple être remplacée par `double Iformes.surface()`. Si rien d'autre n'est changé au code, le compilateur génère une erreur à la ligne 29 car la méthode `surface` ne peut plus être appelée que par une référence de l'interface. Voici ce que la méthode `main()` doit contenir après correction:

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

```

    {
        carre ca = new carre(10.5);
        Iformes test=ca;
        Console.WriteLine(test.surface());
        perimetre(ca);
    }

```

5.12. Les appels de constructeurs.

Nous avons vu à plusieurs reprises à travers les différents exercices la façon dont les différents constructeurs sont appelés.

Par défaut, un constructeur normal est appelé à chaque création d'une instance de la classe. Si plusieurs constructeurs existent, les appels suivent les mêmes règles que pour les appels des méthodes surchargées. Pour le constructeur statique, l'exécution est effectuée avant le premier appel à un des membres de la classe.

On ne peut appeler explicitement un constructeur que dans un des cas suivants:

- appel au constructeur de la classe de base dont on hérite.
- appel à un autre constructeur de la même classe

Reprenons l'exemple que l'on a traité sur la classe carre héritant de la classe rectangle:

```

4      class Rectangle
5      {
6          protected float longueur;
7          protected float largeur;
8          public Rectangle(float largeur, float longueur)
9          {
10             this.largeur = largeur;
11             this.longueur = longueur;
12         }
13         public float surface()
14         {
15             return largeur*longueur;
16         }
17     }
18     class Carre:Rectangle
19     {
20         public Carre(float largeur):base(largeur,0)
21         {
22
23         }
24         public float surface()
25         {
26             return largeur*largeur;
27         }
28     }

```

Remarques:

Nous retrouvons à la ligne 20 l'appel à un constructeur de la classe de base sous la forme

public Carre(float largeur):base(largeur,0)

Comme le langage C# prévoit l'héritage d'une seule classe de base, l'appel à son constructeur a été fortement simplifié puisque nous retrouvons *base (largeur,0)*.

Si dans notre exemple nous créons un constructeur par défaut dans la classe 'carre', nous pourrions retrouver un appel de constructeur sous la forme suivante:

```

1      public Carre(float largeur):base(largeur,0)
2      {

```

```

3
4 }
5 public Carre():this(10)
6 {
7 }

```

Nous retrouverons donc uniquement *base* ou *this* comme mots clefs autorisés pour l'appel à un autre constructeur. Tout manquement à cette règle provoquerait une erreur lors de la compilation.

5.13. Les destructeurs et Dispose().

Pour bien comprendre l'utilité d'un destructeur, il est important de se fixer de nouveau les idées sur la façon dont les objets sont créés et détruits sous .NET.

La seule façon d'instancier une classe, c-à-d de créer un nouvel objet est d'utiliser l'opérateur new qui se présente sous la syntaxe suivante:

classe x = new classe (); classe doit être remplacé par le nom de la classe que nous souhaitons instancier et les parenthèses peuvent contenir des paramètres suivant les constructeurs que nous désirons implicitement appeler si nous les avons créés dans notre classe.

Dans le langage C++, un objet pouvait être créé sans utiliser l'opérateur new. Celui-ci était détruit en fonction du fait que ce soit un objet local défini dans une fonction ou un bloc d'instructions ou un objet global défini avant la fonction main. Un objet local était détruit lorsque l'on quittait la fonction ou le bloc et un objet global détruit à la sortie du programme. Pour rappel, un objet local était créé sur la pile tandis qu'un objet global sur le tas.

Dans le langage C++, l'opérateur new permettait de réaliser de l'allocation dynamique, celui-ci étant lié à l'opérateur delete permettant de libérer l'espace mémoire occupé par l'objet.

Dans le langage C#, la création d'un objet doit se faire obligatoirement par l'opérateur new. Celui-ci permet de créer un objet sur le tas même si l'utilisation de cet opérateur s'effectue dans une fonction. Il n'existe pas d'opérateur delete.

Tout comme pour l'opérateur new en C++ qui nous obligeait à déclarer un pointeur capable de contenir l'adresse renvoyée, le C# nous oblige à utiliser une référence capable de référencer l'objet créé. Tandis que la référence sera placée sur la pile, l'objet est placé sur le tas.

Lorsque un objet n'est plus référencé, ce n'est plus le programmeur qui libère l'espace mémoire occupé par cet objet mais c'est un des éléments du runtime que l'on appelle Garbage Collector (ramasse miettes). Le rôle de ce ramasse miettes est d'assurer la gestion de la mémoire et d'enlever le cas échéant des objets n'étant plus référencés.

Pour bien comprendre ce concept, reprenons l'exemple suivant:

```

1 class complexe
2 {
3     float reel;
4     float image;
5     public static int compteur=0;
6     public complexe(float reel, float image)
7     {
8         this.reel=reel;
9         this.image=image;
10        compteur++;
11    }
12    ~complexe()
13    {
14        compteur --;
15    }
16 }
17
18 class Class1

```

```

19     {
20         static void Main(string[] args)
21         {
22             complexe x = new complexe(1.2F,2.5F);
23             complexe y = new complexe(1.2F,2.5F);
24             creation();
25         }
26         static void creation()
27         {
28             complexe a = new complexe(10.0F,15.0F);
29             complexe b = new complexe(5.0F,4.5F);
30             complexe c = new complexe(1.1F,2.2F);
31         }
32     }

```

L'idée générale de ce code est d'utiliser un compteur d'instance qui est un membre statique de la classe; ce compteur est incrémenté dans le constructeur et décrémenté dans le destructeur. Le destructeur est une méthode analogue au constructeur mais appelée implicitement lors de la destruction de l'objet. Elle porte le même nom que celui de la classe, n'a pas de type de retour ni de paramètre.

Dans la fonction *creation()*, nous créons trois objets de la classe *complexe* et nous initialisons trois références en correspondance avec chacun de ces objets; les références étant créées sur la pile, elles seront détruites lorsque l'on sortira de la fonction.

Dans la fonction *Main*, nous créons deux objets et ensuite nous appelons la fonction *creation()*.

La question que l'on peut se poser est de savoir combien d'objet resteront présents sur le tas lorsque, après l'appel à la fonction *creation()*, on affichera la valeur du membre statique *compteur*. Comme c'est le GC qui s'occupe de gérer le tas, il y a de forte chance qu'aucun objet ne sera encore détruit malgré le fait que parmi les cinq, trois de ces objets ne sont plus référencés. Si nous ajoutons après la ligne 24 un affichage à l'écran du contenu du membre *compteur*, nous aurons la valeur 5.

Question: comment peut on forcer le GC à nettoyer le tas? Il existe en fait une seule solution qui consiste à utiliser la méthode *Collect()*. En fait, cet appel ne doit pas être systématique car il demande des ressources supplémentaires au niveau de votre ordinateur. Le point le plus important est la gestion des ressources non managés tels que par exemple les accès à une base de données. On pourrait supposer placer la libération de telles ressources dans le destructeur mais le destructeur n'est appelé que lors de la destruction de l'objet qui est sous la responsabilité du GC et qui se produira naturellement à un moment non prévisible par le programmeur et bien souvent après que l'on ait quitté l'application. Pour forcer la gestion des ressources non managés, deux solutions sont envisageables:

- Nettoyage complet du tas managé: il existe une classe GC appartenant à l'espace de nom *System* et comprenant la méthode statique *Collect()*. Comme la destruction d'un objet managé provoque l'appel du destructeur, on placera dans ce dernier la libération des ressources non managées par le GC.
- Nettoyage sélectif des ressources non managées liées à un objet managé. Cette possibilité est offerte par l'utilisation de l'interface *IDisposable* qui nous oblige à placer dans les classes dérivées la méthode *Dispose()*.

a) Reprenons le code précédemment développé dans lequel nous utilisons la méthode statique *Collect()*.

```

1     static void Main(string[] args)
2     {
3         complexe x = new complexe(1.2F,2.5F);
4         complexe y = new complexe(1.2F,2.5F);
5         creation();

```

```

6         Console.WriteLine(complexe.compteur);
7         GC.Collect();
8         Console.ReadLine();
9         Console.WriteLine(complexe.compteur);
10    }

```

La ligne 7 permet de demander au GC le nettoyage du tas managé. La ligne 8 permet de mettre votre programme artificiellement en attente pour le nettoyage puisse avoir lieu et la ligne 9 provoquera l'affichage du membre compteur qui contient alors la valeur 2. En effet les trois objets créés dans la fonction ne sont plus référencés et ont donc été supprimés du tas par le GC. Le fait que le compteur se décrémente est bien la preuve que le destructeur est appelé implicitement par le GC lors de la destruction des objets du tas managé.

b) Faisons maintenant hériter notre classe complexe de l'interface *IDisposable*. Cette dérivation nous oblige à implémenter la méthode *Dispose()* qui pourra être appelée par l'objet dont la suppression nécessiterait la gestion de ressources non managées. Qu'en est il de l'objet en lui-même? Est il détruit par lors de l'appel de *Dispose()*?

```

1     class complexe:IDisposable
2     {
3         public float reel;
4         public float image;
5         public static int compteur=0;
6         public complexe(float reel, float image)
7         {
8             this.reel=reel;
9             this.image=image;
10            compteur++;
11        }
12        public void Dispose()
13        {
14        }
15    }
16    ~complexe()
17    {
18        compteur --;
19    }
20 }
21
22 class Class1
23 {
24     static void Main(string[] args)
25     {
26         complexe x = new complexe(1.2F,2.5F);
27         complexe y = new complexe(1.2F,2.5F);
28         creation();
29         y.Dispose();
30         Console.WriteLine(complexe.compteur);
31     }
32     static void creation()
33     {
34         complexe a = new complexe(10.0F,15.0F);
35         complexe b = new complexe(5.0F,4.5F);
36         complexe c = new complexe(1.1F,2.2F);
37     }
38 }

```

L'exécution d'un tel code donne comme affichage 5 c-à-d que le GC n'a pas détruit l'objet puisque le destructeur n'a pas été appelé. Pour s'en convaincre, il suffit d'essayer d'accéder à cet objet après l'appel à la méthode *Dispose()* en ajoutant la ligne de code *Console.WriteLine(y.reel);* L'exécution du code ainsi modifié donnera l'affichage 1.2 pour la partie réelle du complexe y. L'objet n'est donc pas détruit.

A quoi peut donc servir *Dispose()*? A simplement libérer manuellement les ressources non managés tandis que les ressources managés sont du ressort du GC.

La situation est donc la suivante:

- La méthode *Dispose()* est appelée explicitement dans le code et jamais par le GC. Cette méthode doit comprendre la libération des ressources non managés.
- Le destructeur est appelé implicitement par le GC. Ne faut il pas prévoir également la libération des ressources non managées dans le cas où nous oublierions d'appeler la méthode *Dispose()*. Dans ce cas de figure, n'y a-t-il pas un risque que ces méthodes soient appelées toutes les deux? Pour éviter ce problème, nous devons implémenter dans la méthode *Dispose()* l'appel à la méthode *SuppressFinalize(this)* permettant de renseigner au GC que lors de la suppression de l'objet du tas managé, le destructeur ne doit plus être appelé du fait que les ressources non managées ont été libérées.

Voici à quoi devrait donc ressembler notre code:

```
1     class complexe:IDisposable
2     {
3         float reel;
4         float image;
5         public static int compteur=0;
6         public complexe(float reel, float image)
7         {
8             this.reel=reel;
9             this.image=image;
10            compteur++;
11        }
12        public void Dispose()
13        {
14            Dispose(true);
15            GC.SuppressFinalize(this);
16        }
17        protected virtual void Dispose(bool disposing)
18        {
19            //libération des ressources non managées.
20        }
21        ~complexe()
22        {
23            Dispose(false);
24            compteur--;
25        }
26    }
27 }
```

Nous avons créé une méthode *Dispose (bool disposing)* comprenant la libération des méthodes non managées et étant appelable à la fois du destructeur et de la méthode dispose.

Nous remarquons à la ligne 15 l'emploi de la méthode *GC.SuppressFinalize(this)* permettant de renseigner au GC que, dans ce cas, comme les ressources non managées ont été libérées, il n'est plus nécessaire d'appeler le destructeur lors de la destruction de l'objet sur le tas managé.

Pour s'en convaincre, il suffit de modifier le contenu de la fonction *creation ()* en y incluant les lignes de code suivantes:

```

1      static void creation()
2      {
3          complexe a = new complexe(10.0F,15.0F);
4          complexe b = new complexe(5.0F,4.5F);
5          complexe c = new complexe(1.1F,2.2F);
6          c.Dispose();
7          b.Dispose();
8      }

```

Si nous appelons la méthode *GC.Collect()* dans la fonction *Main()*, que va-t-il se passer lors de l'affichage du membre compteur qui n'est que décrémenté dans le destructeur? Voici le contenu de la fonction *Main()*:

```

1      static void Main(string[] args)
2      {
3          complexe x = new complexe(1.2F,2.5F);
4          complexe y = new complexe(1.2F,2.5F);
5          creation();
6          GC.Collect();
7          Console.ReadLine();
8          Console.WriteLine(complexe.compteur);
9      }
10

```

Si nous comptons le nombre d'objets créés et le nombre d'objets que le GC va détruire, nous obtenons:

5 objets créés: 2 dans la *Main()* et 3 dans la fonction *creation()*.

3 objets détruits puisque les 3 objets créés dans la fonction *creation()* ne sont plus référencés.

Malgré tout, l'affichage du membre compteur donne la valeur 4. En effet, pour les références c et b, la méthode *Dispose()* a été appelée et de ce fait *SuppressFinalize()* également. Le GC n'a donc pas appelé le destructeur des objets associés aux références c et b.

Nous pouvons finalement aborder l'aspect suivant: que se passe-t-il si un des membres de la classe est lui-même un objet d'une autre classe présentant elle-même une méthode *Dispose()*? Cette méthode doit être appelée explicitement lors de l'appel à notre propre méthode *Dispose()* mais pas lorsqu'il s'agit de l'appel de notre destructeur puisque cet objet est lui-même géré et que donc son destructeur sera appelé. Nous obtiendrons alors le code suivant:

```

1      protected virtual void Dispose(bool disposing)
2      {
3          if (disposing == true)
4          {
5              //appel d'autres méthodes Dispose()
6              //d'objet gérés appartenant à la classe
7          }
8          //libération des ressources non gérées.
9      }

```

Nous ajouterons simplement que comme la méthode *Dispose()* peut être appelée à plusieurs reprises, l'objet n'étant pas été détruit, il est important d'ajouter un booléen comme membre de la classe permettant ainsi d'effectuer un test. Voici le code:

```

1      class complexe:IDisposable
2      {
3          float reel;
4          float image;
5          bool disposed;

```

```

6         public static int compteur=0;
7         public complexe(float reel, float image)
8         {
9             this.reel=reel;
10            this.image=image;
11            compteur++;
12            disposed=false;
13        }
14        public void Dispose()
15        {
16            if (!disposed)
17            {
18                Dispose(true);
19                GC.SuppressFinalize(this);
20                disposed=true;
21            }
22        }
23        protected virtual void Dispose(bool disposing)
24        {
25            //libération des ressources non managées.
26        }
27        ~complexe()
28        {
29            Dispose (false);
30            compteur --;
31        }
32    }

```

Reprenons un exemple complet dans lequel nous utiliserons notamment une classe Image proposant sa méthode *Dispose()*. Il sera intéressant de disposer de l'analyseur de performances du gestionnaire de tâches pour voir comment évolue la mémoire

```

1     class GestionImage:IDisposable
2     {
3         Image picture = null;
4         protected bool disposed=false;
5         public string Picture
6         {
7             set
8             {
9                 picture=Image.FromFile(value);
10            }
11        }
12        public GestionImage()
13        {
14            Console.WriteLine("appel constructeur");
15        }
16        public void Dispose()
17        {
18            Console.WriteLine ("Appel explicite de libération");
19            Dispose(true);
20            GC.SuppressFinalize(this);
21        }
22        }
23        protected virtual void Dispose(bool disposing)
24        {
25            if (!disposed)
26            {
27                Console.WriteLine("ressources pas encore
28                libérées");
                if (disposing == true)

```

```

29         {
30             Console.WriteLine("libération ressources
31                                 managées");
32             if (picture != null)
33             {
34                 picture.Dispose();
35                 picture=null;
36             }
37             Console.WriteLine("libération ressources non
38                                 managées");
39             disposed=true;
40         }
41     else
42     {
43         Console.WriteLine("ressources déjà libérées");
44     }
45     ~GestionImage()
46     {
47         Console.WriteLine("appel destructeur");
48         Dispose (false);
49     }
50 }
51
52 class Class1
53 {
54     static void Main(string[] args)
55     {
56         GestionImage temp1=new GestionImage();
57         temp1.Picture="c:\\image.jpg";
58         GestionImage temp2=new GestionImage();
59         temp2.Picture="c:\\image.jpg";
60         GestionImage temp3=new GestionImage();
61         temp3.Picture="c:\\image.jpg";
62         Console.ReadLine();
63         temp1.Dispose();
64         Console.ReadLine();
65         Console.WriteLine("fin du programme");
66     }
67 }
68 }

```

De la ligne 56 à la ligne 61, nous allons créer des objets de type image. Le fichier qui nous sert en test fait 656Ko, ce qui nous permet d'avoir un impact sur les jauges présentées dans l'analyseur de performances. L'évolution des jauges n'est pas nécessairement proportionnelle à la taille du fichier. Avant toute exécution du code, l'utilisation du fichier d'échange était de 223Mo. Lorsque nous arrivons au premier *Readline()* , nous obtenons la valeur suivante: 246Mo.

Au deuxième *ReadLine()*, nous obtenons 240Mo et à la sortie du programme, 223Mo.

L'affichage après exécution est le suivant:

```

appel constructeur
appel constructeur
appel constructeur

```

```

Appel explicite de libération
ressources pas encore libérées
libération ressources managées
libération ressources non managées

```

fin du programme
appel destructeur
ressources pas encore libérées
libération ressources non managées
appel destructeur
ressources pas encore libérées
libération ressources non managées

5.14. Les délégués.

Dans le langage C et C++ nous avons la possibilité de déclarer des pointeurs de fonctions permettant de contenir une adresse correspondant à une fonction à exécuter. En C#, les pointeurs de fonctions ont été remplacés par les délégués. Ils interviennent lorsque l'on désire passer des méthodes à d'autres méthodes. Nous pouvons penser à la méthode de la classe `Thread` permettant de démarrer un thread et dont la méthode `Thread.Start()` a besoin d'un paramètre correspondant à la méthode qui doit être invoquée par le thread.

On peut voir les délégués comme étant un nouveau type en C#. En effet, l'utilisation des délégués nécessite une déclaration de type, la création de références et des instantiations.

Pour bien comprendre l'utilisation de références, nous allons considérer l'exemple suivant: soit une méthode devant assurer le tri d'un tableau d'entiers suivant la méthode du tri à bulles.

```
1     class Class1
2     {
3         static void Main(string[] args)
4         {
5             int [] tab = {2,7,1,4,3,8,6,5,10,9};
6             tri(tab);
7             for (int i=0; i<tab.Length;i++)
8             {
9                 Console.WriteLine(tab[i]);
10            }
11        }
12        static void tri(int [] tab)
13        {
14            for (int i=0; i< tab.Length;i++)
15            {
16                for (int j=i+1;j<tab.Length;j++)
17                {
18                    if (tab[j]<tab[i])
19                    {
20                        int temp = tab[i];
21                        tab[i]=tab[j];
22                        tab[j]=temp;
23                    }
24                }
25            }
26        }
27    }
```

Imaginons maintenant que l'on désire trier un tableau d'objets sans avoir à réécrire la fonction assurant le tri. Nous devons naturellement changer les éléments suivants dans la fonction:

- A la ligne 18, nous retrouvons une comparaison d'entiers. Il faudrait remplacer le code par une comparaison d'objets qu'on désire trier.
- Aux lignes 20, 21 et 22 nous permutons deux entiers; il faut à cet endroit modifier le code pour que l'opération porte sur des objets.

L'idée est de remplacer l'opérateur de comparaison par une méthode capable de comparer deux des objets que l'on désire trier et de renvoyer un booléen. Pour que nous puissions utiliser n'importe quel type d'objets, l'idée est de pouvoir passer la méthode en paramètre, méthode adaptée aux objets à trier. C'est dans ce cadre que l'utilisation des délégués est intéressante.

La déclaration du type délégué sera sous la forme suivante:

```
delegate bool Compare(Object x, Object y);
```

delegate	Mot clef permettant de déclarer un type délégué.
bool	C'est le type de retour de la fonction vers laquelle le l'instance du délégué pointera.
Compare	Nom du type délégué.
(object x, object y)	Liste des paramètres de la fonction vers laquelle l'instance du délégué pointera. (*)

(*) Sachant qu'en C# tout est classe, que ces classes héritent d'une classe de base qui est object et que toute instance d'une classe de base peut pointer vers un objet d'une classe dérivée, on comprend que si une instance du délégué 'Compare' peut pointer vers n'importe quelle méthode capable de trier un type d'objet donné, on doit utiliser comme type de paramètre une référence à la classe de base object.

```

1  delegate bool Compare(object x, object y);
2
3  class Class1
4  {
5      static void Main(string[] args)
6      {
7          int [] tab = {2,7,1,4,3,8,6,5,10,9};
8          Compare comp = new Compare(CompInt);
9          tri(tab,comp);
10         for (int i=0; i<tab.Length;i++)
11         {
12             Console.WriteLine(tab[i]);
13         }
14     }
15     static void tri(int [] tab,Compare comp)
16     {
17         for (int i=0; i< tab.Length;i++)
18         {
19             for (int j=i+1;j<tab.Length;j++)
20             {
21                 if (comp(tab[j],tab[i]))
22                 {
23                     int temp = tab[i];
24                     tab[i]=tab[j];
25                     tab[j]=temp;
26                 }
27             }

```

```

28         }
29     }
30     static bool CompInt(object x, object y)
31     {
32         return ((int)x<(int)y);
33     }
34 }

```

Commentaires:

A partir de la ligne 30, nous retrouvons la méthode capable de comparer deux entiers. Il est important de conserver des paramètres de type *object* pour rester en conformité avec la référence du type délégué '*Compare*' avec lequel nous travaillerons. A la ligne 20, nous retrouvons la comparaison des deux entiers en nous obligeant à effectuer un transtypage entre les références de la classe de base et les entiers correspondant au type natif de l'objet à comparer.

A la ligne 8, nous retrouvons la créations d'une référence sur le type délégué *Compare* et la créations d'une instance de ce délégué par l'opérateur *new* et le passage en paramètre du nom de la méthode vers laquelle la référence doit pointer.

A la ligne 15, nous faisons passer une fonction en paramètre par l'intermédiaire d'une référence au délégué *Compare*.

Bien que cet exemple soit fonctionnel, il se limite à des tableaux d'entiers puisque la fonction de tri comprend encore comme premier paramètre une référence sur un tableau d'entiers et la permutation se base encore sur une variable temporaire qui est de type entier. Nous allons adapter notre code sur base de nos propres objets correspondant à une classe client.

```

1     class client
2     {
3         string Nom;
4         double dettes;
5         public double Dettes
6         {
7             set
8             {
9                 dettes=value;
10            }
11            get
12            {
13                return dettes;
14            }
15        }
16        public client(string Nom, double dettes)
17        {
18            this.Nom=Nom;
19            this.Dettes=dettes;
20        }
21        public client(string Nom):this(Nom,0.0)
22        {
23        }
24    }

```

Le principe de cette classe repose sur la gestion des ardoises au bar des étudiants. Il serait intéressant de pouvoir trier un tableau d'objets de cette classe sur base du montant des dettes. Nous allons donc adapter la méthode de tri sous la forme suivante:

```

1  static bool CompClient(object x, object y)
2  {
3      Client tmp1= (Client)x;
4      Client tmp2= (Client)y;
5      return (tmp1.Dettes<tmp2.Dettes);
6  }

```

Reprenons maintenant l'ensemble du code modifié.

```

1  delegate bool Compare(object x, object y);
2  class Client
3  {
4      public string Nom;
5      double dettes;
6      public double Dettes
7      {
8          set
9          {
10             dettes=value;
11         }
12         get
13         {
14             return dettes;
15         }
16     }
17     public Client(string Nom, double dettes)
18     {
19         this.Nom=Nom;
20         this.Dettes=dettes;
21     }
22     public Client(string Nom):this(Nom,0.0)
23     {
24     }
25 }
26 class Class1
27 {
28     static void Main(string[] args)
29     {
30         Compare comp = new Compare(CompClient);
31         Client [] tab={
32             new Client("Dupond",50.45),
33             new Client("Dubois",10.0),
34             new Client("Dupont",15.50)};
35         tri(tab ,comp);
36         for (int i=0; i<tab.Length;i++)
37         {
38             Console.WriteLine("Le client {0} a {1} euros de
39                 dettes",tab[i].Nom,tab[i].Dettes);
40         }
41     static void tri(object[] tab,Compare comp)
42     {
43         for (int i=0; i< tab.Length;i++)
44         {
45             for (int j=i+1;j<tab.Length;j++)
46             {
47                 if (comp(tab[j],tab[i]))

```

```

48             {
49                 object temp = tab[i];
50                 tab[i]=tab[j];
51                 tab[j]=temp;
52             }
53         }
54     }
55 }
56 static bool CompClient(object x, object y)
57 {
58     Client tmp1= (Client)x;
59     Client tmp2= (Client)y;
60     return (tmp1.Dettes<tmp2.Dettes);
61 }
62
63 }

```

L'exécution du code donne l'affichage suivant:

Le client Dubois a 10 euros de dettes
Le client Dupont a 15,5 euros de dettes
Le client Dupond a 50,45 euros de dettes

Commentaires:

A la ligne 49, nous avons modifié la permutation qui s'effectue maintenant sur base d'objets de la classe *object* et non plus d'objets de type entiers.
A la ligne 41, le premier paramètre est un tableau d'objets de type *object* et non plus un tableau d'entiers.

5.15. Le transtypage défini par l'utilisateur.

Le C# autorise deux types de transtypage: celui implicite et celui explicite. Dans le cas d'un transtypage explicite, celui-ci est marqué explicitement dans le code par le type de données de destination entre parenthèses.

Exemple:

```

int a = 10;
long b = a; // transtypage implicite
short c = (short) a; //transtypage explicite.

```

Pour nos propres classes, nous pouvons définir des transtypage de façon analogue à la surcharge des opérateurs en utilisant les mots clefs *implicit* ou *explicit*. Reprenons notre exemple du paragraphe précédent en nous intéressant à la classe *Client*. Nous pourrions envisager la syntaxe suivante où l'on convertirait un objet de cette classe vers un entier qui correspondrait par exemple à ses dettes.

```

Client x = new x ("Dupond");
int dette = (int) x;

```

```

1     delegate bool Compare(object x, object y);

```

```

2     class Client
3     {
4         public string Nom;
5         double dettes;
6         public static explicit operator double(Client x)
7         {
8             return x.Dettes;
9         }
10        public double Dettes
11        {
12            set
13            {
14                dettes=value;
15            }
16            get
17            {
18                return dettes;
19            }
20        }
21        public Client(string Nom, double dettes)
22        {
23            this.Nom=Nom;
24            this.Dettes=dettes;
25        }
26        public Client(string Nom):this(Nom,0.0)
27        {
28        }
29    }
30    class Class1
31    {
32        static void Main(string[] args)
33        {
34            Client x = new Client("Dupond",10.50);
35            double dette = (double)x;
36            Console.WriteLine("Le client {0} a {1} euros de
37
38                dettes",x.Nom, (double)x);
39        }

```

Commentaires:

La ligne 6 reprend la syntaxe permettant la mise en place de notre transtypage explicite:
public static explicit operator double(Client x)
public static explicit operator sont les mots clefs,
double indique le type résultant du transtypage
Client x indique le type et la référence devant être transtypés.

6. La gestion des erreurs et des exceptions.

6.1. Introduction.

Jusqu'à présent, les différentes exceptions qui étaient levées dans nos codes d'exemples étaient récupérées par le système d'exploitation et provoquaient l'arrêt de notre programme

avec l'affichage d'un message d'erreur. Il est plus adéquat de capturer autant que possible ces exceptions afin de contrôler le bon déroulement de votre application. Vous pouvez également générer des exceptions par les classes mises à votre disposition héritant de *System.Exception* mais également créer vos propres classes d'exception par la dérivation de *ApplicationException*.

6.2. Capture d'exception.

La capture des exceptions en C# peut s'effectuer en subdivisant la partie de votre code en trois blocs caractérisés par les mots clefs *try*, *catch* et *finally*.

- Le bloc *try* correspond au bloc qui contient les opérations normales pouvant être à l'origine de l'erreur.
- Le bloc *catch* correspond au bloc qui contient le code à exécuter en cas d'erreur.
- Le bloc *finally* correspond au bloc contenant le code permettant de nettoyer les ressources ou d'effectuer toute action que vous souhaitez effectuer à la fin d'un bloc *try* ou *catch*. Le contenu du bloc *finally* est exécuté qu'il y ait une condition d'erreur ou pas. Ce bloc est optionnel.

Considérons l'exemple d'une division par zéro.

```
1     class Class1
2     {
3         static void Main(string[] args)
4         {
5             int a=20;
6             int b=0,c;
7             try
8             {
9                 c=a/b;
10            }
11            catch(System.DivideByZeroException e)
12            {
13                Console.WriteLine("Divison par zéro");
14            }
15            Console.WriteLine("Fin du programme");
16        }
17    }
18 }
```

Nous pouvons spécifier des blocs *catch* multiples pour permettre des types différents d'exceptions. Une complication peut surgir du fait que les exceptions forment une hiérarchie d'objets et de ce fait, une exception particulière peut trouver une correspondance avec plus qu'un seul bloc *catch*. Etant donné que c'est le premier bloc *catch* rencontré correspondant à l'exception générée qui est exécuté, il est important de placer en premier les blocs *catch* correspondant à des exceptions plus spécifiques et ensuite de placer les blocs *catch* correspondant aux exceptions plus générales.

Nous pouvons également générer les exceptions avec l'emploi de *throw*. Reprenons le code précédent et adaptons-le à la lumière de ce qui vient d'être dit.

```
1     static void Main(string[] args)
2     {
```

```

3      int a=20;
4      int b=0,c;
5      try
6      {
7          if (b == 0) throw new
            DivideByZeroException("division par zéro");
8          else c=a/b;
9      }
10     catch(System.DivideByZeroException e)
11     {
12         Console.WriteLine("Cause d'exception: "+e.Message);
13     }
14     catch (System.Exception e)
15     {
16         Console.WriteLine("Autre exception: "+e.Message);
17     }
18     Console.WriteLine("Fin du programme");
19 }

```

Commentaires:

La ligne 14 comprend un bloc *catch* qui permettra d'intercepter les autres exceptions qui ne sont pas prises en charge par un bloc *catch* spécifique précédent.

La ligne 7 permet de générer soi même une exception dans le code.

Les lignes 12 et 14 mettent en évidence l'accès à une propriété de la classe de base *Exception* qui est *Message* permettant d'afficher le texte natif qui décrit la condition d'erreur. Nous pouvons retrouver les autres propriétés suivantes:

<i>HelpLink</i>	Lien à un fichier d'aide fournissant plus d'informations sur l'exception
<i>Source</i>	Nom de l'application ou de l'objet ayant causé l'exception
<i>StackTrace</i>	Détails des appels de méthodes sur la pile (pour faciliter la trace de la méthode qui a levé l'exception)
<i>TargetSite</i>	Un objet de réflexion .NET décrivant la méthode qui a levé l'exception
<i>InnerException</i>	Si cette exception est levée dans un bloc <i>catch</i> , elle contient l'objet "exception" ayant envoyé le code dans ce bloc <i>catch</i>

StackTrace et *TargetSite* sont fournis automatiquement par le runtime .NET si une trace de la pile est disponible. *Source* est toujours remplie par le runtime .NET avec le nom de l'assemblage dans lequel l'exception a été levée (nous pouvons en modifier le contenu). *Message*, *HelpLink* et *InnerException* doivent être remplis par le code qui a levé l'exception.

6.3. Classes d'exceptions définies par l'utilisateur.

Une classe d'exception définie par l'utilisateur doit dériver de la classe de base *ApplicationException*. Imaginons que l'on désire demander l'introduction au clavier d'un nombre compris entre 0 et 255 et que le non respect de cette restriction génère une exception que nous avons créée.

```

1  class Class1
2  {
3      static void Main(string[] args)
4      {
5          int valeur;
6          string Valeur;
7          try
8          {
9              Console.WriteLine("Un nombre entre 0 et 255: ");
10             Valeur = Console.ReadLine();
11             valeur = Convert.ToInt32(Valeur);
12             if ((valeur<0)|| (valeur>255))
13                 throw new OutOfRange("Valeur hors
14                 limite");
15             }
16             catch (OutOfRangeException e)
17             {
18                 Console.WriteLine(e.Message);
19             }
20             catch(System.DivideByZeroException e)
21             {
22                 Console.WriteLine("Cause d'exception:
23                 "+e.Message);
24                 Console.WriteLine(e.TargetSite);
25             }
26             catch (System.Exception e)
27             {
28                 Console.WriteLine("Autre exception:
29                 "+e.Message);
30             }
31             Console.WriteLine("Fin du programme");
32         }
33     }
34 }
35
36 class OutOfRange:ApplicationException
37 {
38     public OutOfRange(string Message):base(Message)
39     {
40     }
41     public OutOfRange(string Message, Exception
42     InnerException):base(Message, InnerException)
43     {
44     }
45 }

```

Commentaires:

A partir de la ligne 31, nous retrouvons la création de notre propre exception sous la forme d'une classe héritant de la classe de base *ApplicationException*. Dans cette classe, nous implémentons deux constructeurs chacun comprenant le message qui devra être fourni en cas d'erreur.

7. Les méthodes anonymes (c# 2.0).

Pour bien comprendre l'utilité des méthodes anonymes, nous allons reprendre l'exemple traitant de l'utilisation des délégués.

```

1  delegate bool Compare(object x, object y);

```

```

2
3 class Class1
4 {
5     static void Main(string[] args)
6     {
7         int [] tab = {2,7,1,4,3,8,6,5,10,9};
8         Compare comp = new Compare(CompInt);
9         tri(tab,comp);
10        for (int i=0; i<tab.Length;i++)
11        {
12            Console.WriteLine(tab[i]);
13        }
14    }
15    static void tri(int [] tab,Compare comp)
16    {
17        for (int i=0; i< tab.Length;i++)
18        {
19            for (int j=i+1;j<tab.Length;j++)
20            {
21                if (comp(tab[j],tab[i]))
22                {
23                    int temp = tab[i];
24                    tab[i]=tab[j];
25                    tab[j]=temp;
26                }
27            }
28        }
29    }
30    static bool CompInt(object x, object y)
31    {
32        return ((int)x<(int)y);
33    }
34 }

```

Pour rappel, l'utilisation d'un délégué nous oblige à respecter trois étapes bien distinctes :

1- La création d'un type délégué.

```
delegate bool Compare(object x, object y);
```

2- La création de la fonction.

```
static bool CompInt(object x, object y)
{
    return ((int)x<(int)y);
}
```

3- L'instanciation du type délégué avec le passage de la fonction en paramètre.

```
Compare comp = new Compare(CompInt);
```

L'utilisation des méthodes anonymes nous permet de pouvoir déclarer la fonction en même temps que le délégué de la façon suivante :

1- delegate bool Compare(object x, object y);

2- Compare comp = new delegate (object x, object y)

```
{
    return ((int)x<(int)y) ;
}
```

```
}
```

Dans cet exemple précédent, nous remarquons que nous avons respecté la signature du délégué lors de la création de la méthode anonyme, comme cela devait être le cas pour l'utilisation des délégués en C#1.0. Ce n'est plus nécessaire actuellement et nous pouvons admettre toute méthode anonyme sous les règles de compatibilité suivantes.

Paramètres :

- La méthode anonyme n'a pas de liste de paramètres et le délégué n'a pas de paramètres déclarés out.
- La liste de paramètres de la méthode anonyme est la même que celle du délégué.

Type de retour :

- Si le délégué retourne void, alors la méthode anonyme ne doit pas contenir l'instruction return, ou seulement l'instruction return sans expression derrière.
- Sinon, les expressions retournées doivent pouvoir être implicitement convertie dans le type de retour du délégué.

Nous pourrions donc imaginer l'exemple suivant :

```
1  delegate void Message(string chaine);
2
3  Message test1 = delegate () {Console.WriteLine("bonjour")};
4
5  Message test2 = delegate(string MaChaine)
6  {
7      Console.WriteLine(MaChaine);
8  }
9
10 Message test1 = delegate {Console.WriteLine("bonjour")};
11
```

En ayant créé un seul type délégué, nous pouvons référencer plusieurs méthodes anonymes différentes. C'est pratique dans la gestion des événements en programmation Windows Forms lorsque nous ne souhaitons pas utiliser les paramètres. Si les parenthèses sont omises, la méthode peut être assignée à n'importe quelle signature.

Une méthode anonyme peut utiliser n'importe quelle variable membre de la classe et elle peut aussi utiliser toute variable locale définie dans la portée de la méthode qui la contient comme si c'était sa propre variable locale. Prenons l'exemple suivant tiré de la documentation :

```
1  delegate int myDelegate();
2
3  class Test
4  {
5      static mydelegate myFunc() //function retournant un délégué
6      {
7          int x=0;
8          myDelegate result = delegate { return ++x;}
9          return result;
10     }
11
12
13
14     static void Main()
15     {
16         myDelegate d = myFunc(); //inférence
17         Console.WriteLine(d());

```

```

18         Console.WriteLine(d());
19         Console.WriteLine(d());
20     }
33 }
34

```

Le résultat de l'exécution du code est 1 2 3, ce qui est assez surprenant si l'on part du principe qu'une variable locale est détruite lorsque l'on sort de la fonction. Pour expliquer cela, il faut évoquer la « durée de vie » de la variable qui est étendue tant qu'il y a au moins un délégué qui le référence.

Le compilateur C# permet également l'inférence pour les délégués c'est-à-dire le fait de pouvoir directement assigner le nom d'une méthode à une référence de type délégué. Nous retrouvons cette utilisation dans l'exemple ci avant à la ligne 16.

Nous pouvons également reprendre l'exemple de début de paragraphe et l'adapter à la lumière de ce qui vient d'être dit :

- 1- La création d'un type délégué.

```
delegate bool Compare(object x, object y);
```

- 2- La création de la fonction.

```

static bool CompInt(object x, object y)
{
    return ((int)x < (int)y);
}

```

- 3- L'instanciation du type délégué avec le nom de la méthode.

```
Compare comp = Compare;
```

8. Les classes génériques (c# 2.0).

8.1. Introduction.

Les génériques permettent de répondre au problème qui se pose aux langages de programmation fortement typés quand on veut manipuler des données sans se soucier de leur type. Avant les génériques, on pouvait utiliser le type object dont tout objet C# hérite. Le framework 1.1 contient d'ailleurs beaucoup de classes manipulant des objets de type object, spécialement dans le namespace System.Collection.

Nous pouvons reprendre notre méthode mise en place permettant d'assurer la comparaison d'objets pour assurer le tri de notre tableau d'entier ou du tableau de clients du bar en fonction des dettes.

```

30     static bool CompInt(object x, object y)
31     {
32         return ((int)x < (int)y);
33     }
34
35     static bool CompClient(object x, object y)
36     {
37         Client tmp1= (Client)x;

```

```

38         Client tmp2= (Client)y;
39         return (tmp1.Dettes<tmp2.Dettes);
40     }
41

```

Afin de ne pas modifier l'appel à la méthode, nous utilisons comme paramètre des références sur la classe object du fait que toute référence d'une classe de base peut référencer n'importe quel objet d'une classe dérivée. Un gros ennui est sans doute les opérations de transtypage (pour les types référence) et de boxing (pour les types valeur) que l'on retrouve dans la méthode de comparaison et qui pour cette dernière sont pénalisants pour la rapidité d'exécution de votre programme.

Les génériques répondent à ces problèmes de contrôle de type et de performance. En effet, ils permettent à une classe, méthode ou autre de garder un typage fort tout en traitant un problème non spécifique à un type particulier.

8.2. Création des types génériques.

Nous reprendrons notre exemple de tri pour les objets de type clients et de type entiers en créant une classe générique. Nous retrouverons la classe Client ainsi qu'une classe Collection dans laquelle nous retrouverons un tableau d'objet qui pourra être de type client mais qui pourrait aussi contenir des entiers.

Pour permettre d'effectuer une comparaison aisée, quelle que soit le type d'objet, nous retrouverons dans chaque classe une méthode CompareTo dont la déclaration se trouve renseignée dans l'interface IComparable.

```

1     class Client:IComparable<Client>
2     {
3         public string Nom;
4         double dettes;
5         public double Dettes
6         {
7             set
8             {
9                 dettes = value;
10            }
11            get
12            {
13                return dettes;
14            }
15        }
16        public Client(string Nom, double dettes)
17        {
18            this.Nom = Nom;
19            this.Dettes = dettes;
20        }
21        public Client(string Nom): this(Nom, 0.0)
22        {
23        }
24        public static explicit operator double(Client x)
25        {
26            return x.Dettes;
27        }
28        public int CompareTo(Client x)

```

```

29     {
30         return (this.Dettes.CompareTo(x.Dettes));
31     }
32 }

```

La classe Client hérite de l'interface générique IComparable, ce qui va nous obliger à créer une méthode CompareTo retournant un entier permettant d'indiquer si l'objet client référencé par « this » est plus petit, plus grand ou égal à l'objet x passé en paramètre. Nous avons convenu de comparer les clients en fonction de leurs dettes, ce que nous retrouvons à la ligne 30 en utilisant la valeur retournée lors de la comparaison des deux membres « Dettes ».

```

1  class Collection<T>
2      where T:IComparable<T>
3  {
4      public List<T> tab = new List<T>();
5
6      public int Compare(T x,T y)
7      {
8          return x.CompareTo(y);
9      }
10
11     public void Add(T x)
12     {
13         tab.Add(x);
14     }
15     public T this[int i]
16     {
17         get
18         {
19             return tab[i];
20         }
21     }
22     public void tri()
23     {
24         T temp;
25         for (int i = 0; i < tab.Count; i++)
26         {
27             for (int j = i + 1; j < tab.Count; j++)
28             {
29                 if (Compare(tab[j], tab[i])<0)
30                 {
31                     temp = tab[i];
32                     tab[i] = tab[j];
33                     tab[j] = temp;
34                 }
35             }
36         }
37     }
38 }

```

Nous pouvons renseigner dans notre classe générique collection le fait que cette classe puisse être liée à des types qui seront passés en paramètre. Nous choisissons une lettre ou un ensemble de lettres séparées par une virgule entre signes < et >.

Dans notre exemple, nous retrouvons la classe collection<T> avec, nouveauté pour le c#, une restriction sur le type de paramètre qui sera dans notre exemple de type IEnumerable<T>, ce qui permettra à notre classe de savoir que toute variable de type T

déclarée dans la classe pourra utiliser la méthode CompareTo. La restriction peut être liée à une seule classe ou plusieurs interfaces.

Dans l'espace de nom `System.Collections.Generic`, il nous est possible d'utiliser une liste d'objets, définie sous la forme d'une classe générique `Liste<T>`. Cette classe nous permet d'ajouter en dynamique des objets de nature diverses qui pourrait donc être des entiers, des clients, des flottants... En voici la syntaxe de déclaration que l'on retrouve à la ligne 4 : `public List<T> tab = new List<T>()`. Nous retrouvons plusieurs méthodes utilisant le type `T` renseigné lors de la déclaration de la classe dont en voici certaines :

-1- La méthode `public void Add(T x)` qui permet d'ajouter une référence sur un type client lorsque nous utilisons la syntaxe suivante

```
CollectionManu<Client> test2 = new Collection<Client>();
test2.Add(new Client("Dupond1", 14.12));
```

Ou un type entier dans le cas suivant

```
Collection<int> test3 = new Collection<int>();
test3.Add(10);
```

Nous remarquons la manière d'instancier la classe qui consiste notamment à renseigner le type utilisé dans la classe, soit dans notre exemple « Client » ou « int ».

-2- L'indexeur `public T this[int i]` qui retourne grace à l'assesseur un des objets de la liste `tab` présent à l'indice `i`.

8.3. Les itérateurs.

Pour mieux comprendre la notion d'itérateurs, il est important de reprendre une notion existant déjà en C#1.1 et que l'on appelle énumérateur. Nous avons envisagé dans le cadre du cours, une nouvelle instruction du C#, connue des programmeurs java mais inconnue des programmeurs c ou c++ : l'instruction `foreach`. Soit l'exemple suivant :

```
string []tab={"Dupond","Dubart","Dupuis"};
foreach (string nom in tab)
{
    Console.WriteLine(nom);
}
```

Nous allons étudier la façon dont nous pourrions implémenter dans notre classe `collection` les méthodes indispensables nous permettant alors de pouvoir utiliser le code suivant :

```
Manu<Client> test2 = new Manu<Client>();

test2.Add(new Client("Dupond1", 14.12));
test2.Add(new Client("Dupond2", 11.12));
test2.Add(new Client("Dupond3", 14.12));
test2.Add(new Client("Dupond4", 13.12));

foreach (Client x in test2)
    Console.WriteLine(x.Nom);
```

Nous devons utiliser les interfaces `IEnumerable` et `IEnumerator`. Ces deux interfaces nous permettent d'intégrer dans notre classe l'utilisation des énumérateurs qui sont des objets permettant de nous déplacer dans un tableau ordonné d'items. Reprenons notre exemple de la classe `Collection` et implémentons dans un premier temps une classe de type `IEnumerator`.

```
1 class MonEnumerateur: IEnumerator<T>
2     {
3         int CurIndex;
4         Collection<T> collection;
5         public MonEnumerateur(Collection<T>collection)
6         {
7             this.collection=collection;
8             CurIndex=-1;
9         }
10        public T Current
11        {
12            get {
13                if (this.CurIndex < collection.tab.Count)
14                    return (collection.tab[this.CurIndex]);
15                else throw new IndexOutOfRangeException();
16            }
17        }
18        public void Dispose() { }
19        object System.Collections.IEnumerator.Current
20        {
21            get
22            {
23                return Current;
24            }
25        }
26        public void Reset() { CurIndex = -1; }
27        public bool MoveNext()
28        {
29            if (CurIndex < collection.tab.Count - 1)
30            {
31                CurIndex++;
32                return true;
33            }
34            else return false;
35        }
36    }
37 }
```

L'implémentation de l'interface `IEnumerator<T>` nous oblige à prévoir les membres suivants dans notre classe :

-1- La propriété `public T Current` nous permettant de récupérer l'item courant. Les informations disponibles sur le site msdn de Microsoft nous renseigne l'obligation de prévoir la version non générique de la propriété `Current` sous la forme suivante :

`object System.Collections.IEnumerator.Current`. L'oubli de cette propriété provoque une erreur à la compilation.

-2- La méthode `public void Reset()` nous permettant de nous positionner sur l'item courant de la position d'origine.

-3- La méthode `public bool MoveNext()` nous permettant de nous déplacer sur l'item suivant correspondant à l'ordre de ceux-ci dans le tableau.

Attention : la classe MonEnumerateur est une classe imbriquée dans la classe Collection.

Une fois cette classe créée, nous pouvons implémenter l'interface IEnumerable dans notre classe collection comme dans notre exemple :

```
1  class Collection<T> :IEnumerable<T>
2      where T:IComparable<T>
3  {
4      public List<T> tab = new List<T>();
5      public IEnumerator<T> GetEnumerator()
6      {
7          return new MonEnumerateur(this);
8      }
9      System.Collections.IEnumerator
10     System.Collections.IEnumerable.GetEnumerator()
11     {
12         return GetEnumerator();
13     }
```

L'implémentation de l'interface IEnumerable<T> nous oblige à prévoir la méthode GetEnumerator dans notre classe. Le rôle de cette méthode est d'instancier notre classe MonEnumerateur et de simplement retourner cette instance. Les informations disponibles sur le site msdn de Microsoft nous renseigne l'obligation de prévoir la version non générique de la méthode GetEnumerator sous la forme suivante :

object System.Collections.IEnumerator.Current. L'oubli de cette propriété provoque une erreur à la compilation.

Il est évident que l'implémentation de ces objets n'est pas très simple. C#, sous sa version 2.0, simplifie la mise en place des itérateurs mais la rend aussi plus puissante.

Il n'est en effet plus nécessaire de prévoir une classe héritant de l'interface IEnumerator. Nous modifierons la méthode GetEnumerator de notre classe Collection en y utilisant le mot clef *yield return* de sorte qu'elle retourne elle-même l'ensemble des différents objet en ayant même la possibilité de les filtrer ou de les trier dans cette méthode.

```
1  class Collection<T> :IEnumerable<T>
2      where T:IComparable<T>
3  {
4      public List<T> tab = new List<T>();
5      public IEnumerator<T> GetEnumerator()
6      {
7          for (int i=0;i<this.tab.Count;i++)
8          {
9              yield return this.tab[i];
10         }
11     }
12 }
```

Nous pouvons également intégrer dans notre collection plusieurs itérateurs, chacun parcourant la collection différemment. Par exemple, nous pourrions parcourir nos clients dans l'ordre inverse sans devoir refaire appel à la méthode de tri.

```
1  class Manu<T>:IEnumerable<T>
2      where T:IComparable<T>
3  {
4      public List<T> tab = new List<T>();
5      public IEnumerator<T> GetEnumerator()
```

```

6      {
7          for (int i=0;i<this.tab.Count;i++)
8          {
9              yield return this.tab[i];
10         }
11     }
12
13     public IEnumerable<T> Reverse
14     {
15         get
16         {
17             for (int i = this.tab.Count-1; i>=0; i--)
18             {
19                 yield return this.tab[i];
20             }
21         }
22     }
23 }

```

L'itérateur doit être mis en place sous la forme d'une propriété dont le type de retour est `IEnumerable<type>`. Dans notre cas, le fait d'avoir une classe générique nous permet l'utilisation du type `<T>`. Pour l'utilisation de cet itérateur dans notre fonction principale, nous retrouverons la syntaxe suivante :

```

1     foreach (Client test in test2.Reverse)
2     {
3         Console.WriteLine(test.Nom);
4     }

```

Il y a quelques limitations sur la façon dont nous pouvons implémenter l'instruction `yield return` dans notre code. Une méthode ou une propriété qui a l'instruction `yield return` ne peut pas contenir une instruction `return` simple parce qu'elle provoquerait une interruption impropre dans l'itération. Nous ne pouvons pas utiliser `yield return` dans une méthode anonyme, ni être placée dans une instruction `try` avec un bloc `catch` (exclus aussi dans le bloc `catch` ou le bloc `finally`).

8.4. Le type *partial*.

La version 1.1 du `c#` nous oblige à placer l'entièreté du code pour une classe donnée dans un seul fichier. `C# 2.0` nous permet de scinder la définition et l'implémentation d'une classe sur de multiples fichiers. Nous pouvons donc placer le code d'une partie de la classe dans un fichier et une autre partie de la classe dans un fichier différent en utilisant juste le mot clef `partial`. Le support du type `partial` est envisageable pour les classes, pour les structures et les interfaces mais il ne peut être utilisé pour les énumérations. Alors que dans le cas du développement en `visual studio.net`, il est toujours délicat de modifier le code généré par l'outil de développement au risque de voir son travail personnel perdu si la classe doit être régénérée, l'utilisation des classes partielles permet d'utiliser la technique du « code-beside class » stockant la partie de code générée par l'outil dans un fichier différent.

Cette technique permet aussi à plusieurs développeurs de travailler sur la même classe sans avoir à vérifier leurs fichiers et ce sans interférence.

Il ne faut pas perdre de vue malgré tout quelques aspects non cumulatifs dans les classes ou structures :

- La visibilité (*public* ou *internal*)
- La classe de base. Une même classe de type *partial* définie dans plusieurs fichiers ne peut se voir hériter dans ces déclarations multiples de classe de bases distinctes.
- Seulement une des classes redéfinie peut implémenter une interface.
- Seulement une des classes redéfinie peut surcharger une méthode abstraite ou une méthode virtuelle.

8.5. Les méthodes anonymes.

En programmation *c#*, l'utilisation des délégués est surtout utilisée pour la gestion des événements en windows forms, les appels asynchrones et la programmation multithreading.

Si nous reprenons le paragraphe 5.14 traitant des délégués dans la version 1.1 du *C#*, nous retrouvons les aspects suivants :

```

1  delegate bool Compare(object x, object y);
2  Compare comp = new Compare(CompInt);
3  static bool CompInt(object x, object y)
4  { //contenu de la méthode CompInt }
```

A la ligne 1, nous retrouvons la création du type délégué.

A la ligne 2, nous retrouvons la création de la méthode vers laquelle le délégué va pointer
A la ligne 3, le délégué est instancié et nous devons lui passer en paramètre le nom de la méthode.

Si nous nous basons sur les nouveautés que nous apporte le *C#* dans sa version 2.0, nous pouvons réécrire le même code en utilisant les méthodes anonymes (sans nom).

```

1  delegate bool Compare(object x, object y);
2  Compare comp = delegate (object x, object y);
3  { //contenu de la méthode CompInt }
```

Dans notre exemple, la méthode anonyme a la même signature que le délégué devant la référencer mais ce n'est pas toujours nécessaire. Pour qu'une méthode anonyme puisse être acceptée, les règles de compatibilité sont les suivantes :

Pour les paramètres :

- La méthode anonyme n'a pas de liste de paramètres et le délégué n'a pas de paramètres déclarés out.
- La liste de paramètres de la méthode anonyme est la même que celle du délégué.

Pour le type de retour :

- Si le délégué retourne void, alors la méthode anonyme ne doit pas contenir l'instruction return, ou seulement l'instruction return sans expression derrière.
- Sinon, les expressions retournées doivent pouvoir être implicitement convertie dans le type de retour du délégué.

Les méthodes anonymes peuvent aussi avoir accès aux variables locales externes dans le scope auquel la méthode est liée.

Les méthodes anonymes peuvent aussi avoir des paramètres de type générique tout comme pour les autres méthodes. Elle peut aussi utiliser des types génériques définis dans le scope de la classe

8.6. Les classes statiques.

Il est fréquent en C# de retrouver des classes statiques qui ne comprennent que des membres statiques ou des méthodes statiques. Dans ce cas, une instantiation de ces classes n'a pas de raison d'être. Pour que nous ne puissions pas instancier de telles classes, la seule solution en C#1.1 est de rendre le constructeur par défaut privé. Sans constructeurs publics, il n'est pas possible d'instancier une classe de ce type. C# 2.0 supporte maintenant les classes statiques en permettant l'ajout du mot clef `static` dans la définition de la classe `public static class MaClasse{ }`. Une telle classe ne peut avoir de méthodes instanciables et ne peut servir comme classe de base dans un héritage.

8.7. Le qualifieur d'espace de nom global ::

Il est possible d'utiliser un espace de nom imbriqué qui a le même nom qu'un des espaces de noms globaux. Dans notre exemple, nous retrouvons imbriqué notre espace de nom `System`. Dans un tel cas, le compilateur a des difficultés à résoudre la référence à cet espace de nom global et ne parviendra pas à compiler la ligne 10.

```
1 namespace ConsoleApplication2
2 {
3
4     namespace System
5     {
6         class Program
7         {
8             static void Main(string[] args)
9             {
10                System.Console.WriteLine("bonjour");
11            }
12        }
13    }
14 }
```

Le C# 2.0 permet l'utilisation du qualifieur d'espace de nom global `::` pour indiquer au compilateur qu'il doit démarrer sa recherche dans le scope global. Nous pouvons appliquer le qualifieur `::` à la fois aux espaces de noms et aux types. Le qualifieur sera précédé du mot clef `global` ou de tout alias. Dans notre exemple, la ligne 10 pourra donc être remplacée par la syntaxe suivante :

```
global::System.Console.WriteLine("bonjour");
```

```
1 using test=System ;
2 namespace ConsoleApplication2
3 {
4     namespace System
5     {
6         class Program
```

```

7         {
8             static void Main(string[] args)
9             {
10                test::Console.WriteLine("bonjour");
11            }
12        }
13    }
14 }

```

Voici un autre exemple mettant en évidence l'utilisation du qualifieur pour les types :

```

1 namespace MyApp
2 {
3     class MyClass
4     {
5         static void Main()
6         {
7         }
8         public void MyMethod()
9         {
10            global::MyClass obj = new global::MyClass();
11            obj.MyMethod();
12        }
13    }
14 }
15 public class MyClass
16 {
17     public void MyMethod()
18     {
19         global::System.Console.WriteLine("Hello");
20     }
21 }

```

8.8. Nullable types.

Le problème rencontré avec la version C#1.1 pour les types valeur provenait des difficultés lors de liaisons avec de sbases de données pour ces variables puissent contenir la valeur null qui est en fait réservée pour les types références ne référénçant aucun objet. Ce nouveau type est consrduit en utilisant le '?'. Si nous désirons créer une variable de type entière capable de contenir la valeur null, nous aurons alors la syntaxe suivante :

```
int? x = null ;
```

Un tel type est un fait une instance de la structure *System.Nullable* et va, en plus de fournir la fonctionnalité habituelle du type int dans notre exemple, permettre à la variable de contenir également la valeur *null*. La structure offre également les deux propriétés :

- *HasValue* qui retournera true si la variable contient une valeur et false si le contenu est null
- *Value* qui retournera la valeur assignée à la variable et dans le cas contraire, une exception de type *System.InvalidOperationException* sera levée.

```

1 namespace ConsoleApplication4
2 {
3     class NullableExample
4     {

```

```

5         static void Main()
6         {
7             int? num = null;
8             if (num.HasValue == true)
9             {
10                System.Console.WriteLine("num = " + num.Value);
11            }
12            else
13            {
14                System.Console.WriteLine("num = Null");
15            }
16
17            int y = num.GetValueOrDefault();
18
19            try
20            {
21                y = num.Value;
22            }
23            catch (System.InvalidOperationException e)
24            {
25                System.Console.WriteLine(e.Message);
26            }
27        }
28    }
29 }

```

Nous pouvons nous poser la question de savoir ce qui se passe lorsque nous voulons utiliser des opérateurs arithmétiques avec ces types ou également vouloir utiliser des opérateurs logiques lorsque les variables sont de type *bool?* .

Les opérateurs prédéfinis, unaires et binaires, ainsi que les surcharges d'opérateurs qui existent pour des opérandes de type valeur peuvent être aussi utilisés pour des *nullable type*. Ces opérateurs produisent une valeur null si les opérandes ont comme valeur *null*; autrement, l'opérateur utilise la valeur contenue pour calculer le résultat. Lorsque des comparaisons sont effectuées sur ces types et que l'une ou l'autre des valeurs est nulle, le résultat de la comparaison sera toujours *false*.

Une variable de type *bool?* Peut contenir trois valeurs différentes : *true*, *false* et *null*. De ce fait, elles ne peuvent être utilisées dans des instructions conditionnelles de type *if*, *for* ou *while*. Nous reprenons une table de vérité pour les opérateurs logiques comprenant des opérandes de ce type :

x	y	x&y	x y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	true	true

Nous retrouvons également pour nouveaux types, le nouvel opérateur binaire *??*. Cet opérateur a comme opérande de gauche une variable de type 'nullable' et renverra sa valeur si elle n'est pas nulle tandis qu'il renverra le contenu de l'opérande de droite dans le cas contraire. Prenons l'exemple suivant :

```

1     static void Main(string[] args)
2     {

```

```
3     int? x = null;
4     int b = 20;
5     int y = x ?? b;
6     Console.WriteLine("Contenu de y:" + y.ToString());
7 }
```

Dans cet exemple, l'exécution du code correspondant provoquera comme affichage :
contenu de y: 20