

Support de cours
Architecture des SI I

Chapitre 2

Langage C#

SOMMAIRE

Introduction à C#	3
Espace de noms	4
Types	4
Membres d'un type	6
Encapsulation	9
Types génériques	10
Collections	11
Héritage	13
Polymorphisme	15
Instanciation.....	17
Passage d'un paramètre	19
Inférence de type	20
Type DateTime	21
Méthodes d'extensions.....	21
Fonctions anonymes	22
Expressions lambda	23
LINQ.....	24
Méthodes d'extensions du LINQ.....	24
LINQ intégré	27
Compléments.....	28
Exceptions.....	28
Conversion de type	30
Complément types	31
Complément commentaires	32
Complément membres d'un type	33
Complément collections.....	36
Complément héritage.....	37
Complément polymorphisme	38
Complément passage d'un paramètre	38
Complément espace de noms	40
Complément chaînes de caractères	41
Surcharge d'opérateurs.....	41
Point d'entrée d'un assembly.....	41
Bonnes pratiques de nommage.....	42
Structures de contrôle	43
Références	46

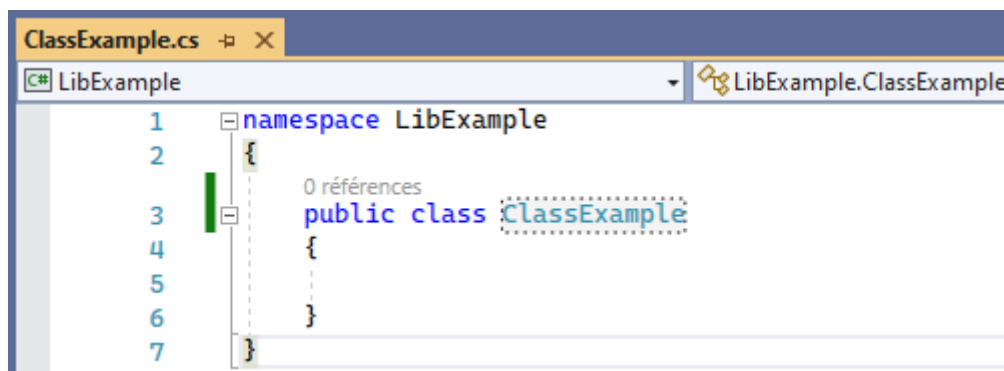
Introduction à C#

C# est un langage introduit par Microsoft en 2002. C'est un langage orienté objet avec un typage statique fort, une syntaxe héritée du C/C++ et une philosophie très proche de Java. Il est sensible à la casse. C'est le langage conseillé par Microsoft pour tout nouveau projet .NET. Le langage lui-même a évolué au cours du temps et voici un tableau des différentes versions

Année	Version	Bibliothèque	Principal changement
2002	1.0	.NET framework 1.0 et 1.1	
2005	2.0	.NET framework 2.0	généricité ajoutée à C# et au framework
2008	3.0	.NET framework 3.5	LINQ (Language integrated queries)
2010	4.0	.NET framework 4.0	types dynamiques
2012	5.0	.NET framework 4.5	méthodes asynchrones
2015	6.0	.NET framework 4.6	version pour Linux
2016	7.0	.NET framework >= 4.5	Tuples, fonctions locales
2019	8.0	.NET standard >=2.1 et .NET Core >=3.0	Membre ReadOnly, opérateur d'assignation de fusion
2020	9.0	.NET 5.0	Vérification de nullité, instructions de niveau racine
2021	10.0	.NET 6.0	Directives d'utilisation globale, modèles de propriété étendus

Un fichier de code source écrit en langage C# porte l'extension **cs**.

Voici un exemple de contenu d'un fichier C#



En C# et comme en C/C++ ou Java les commentaires se font de la même manière

```
// Je suis un commentaire sur une seule ligne.
```

```
/* Je suis un commentaire sur plusieurs lignes.
   J'apparais grisé dans les éditeurs offrant une coloration syntaxique */
```

Espace de noms

Dans le précédent exemple on remarque la présence du mot clé **namespace**. Ce mot clé permet de spécifier l'espace de noms des entités créées dans le bloc entre les accolades. Dans ce cas l'entité **ClassExample** réside dans l'espace de noms **LibExample**. Le nom d'une entité avec son espace de noms permet de spécifier son nom complet, ainsi le nom complet de l'entité est « LibExample.ClassExample ».

Lors de l'utilisation de cette entité on peut se servir de son nom complet par exemple

```
var instance = new LibExample.ClassExample();
```

Pour ne pas allonger l'écriture il est possible d'utiliser le mot clé **using**. Dans ce cas on n'a pas besoin d'inclure l'espace de noms et le nom de l'entité suffit

```
using LibExample;
```

```
var instance = new ClassExample();
```

L'inclusion du mot clé **using** n'est visible que dans le périmètre d'un fichier, il est possible faire autrement en utilisant les espaces de noms globaux et ainsi avoir une portée plus large. Pour déclarer un using global il suffit d'utiliser le mot clé **global** comme ceci

```
global using LibExample;
```

Les espaces de noms globaux n'ont d'effet que sur un projet. En général on crée un fichier nommé **GlobalUsings** qui contiendra les espaces de noms globaux de notre projet.

Par défaut et selon le type de projet on a des espaces de noms globaux déclarés implicitement. Voici les espaces de noms globaux d'un projet type *Console*

```
global using System;  
global using System.Collections.Generic;  
global using System.IO;  
global using System.Linq;  
global using System.Net.Http;  
global using System.Threading;  
global using System.Threading.Tasks;
```

Types

En .NET pour structurer les données ou créer des contrats on a besoin des types. Le concept de type est plus large que le concept de classe en y ajoutant d'autres. Dans cette partie on va décrire les différents genres des types utilisés en .NET. Afin de ne mettre que le nécessaire, les exemples exposés seront vides.

On commence par le genre de type la plus connu de l'orienté objet qui est la classe. Une classe est comme dans d'autres langages (Java spécialement) est un regroupement de données avec des fonctions (ou méthodes) pour manipuler ces données. Pour déclarer une classe on doit utiliser le mot clé **class** en spécifiant le nom de cette classe, par exemple

```
1 référence
public class ClassExample
{
}
```

Le deuxième genre de type utilisé en .NET est la structure. Pour déclarer une structure on doit utiliser le mot clé **struct** en spécifiant le nom de cette structure, par exemple

```
0 références
public struct StructExample
{
}
```

Il existe des différences (qu'on va les mentionner dans d'autres parties) entre les structures et les classes. La différence la plus importante est au niveau de la gestion de la mémoire. En effet les structures résident au niveau de la pile alors que les classes résident au niveau du tas. La libération de la mémoire de la pile se fait rapidement en utilisant le mécanisme de la libération de la pile. Pour les classes la libération est gérée par le ramasse-miettes (Garbage Collector) qui est relativement lente. En général les structures sont utilisées dans le cas où on cherche la performance.

L'énumération est un autre genre de type, elle sert pour déclarer un ensemble de valeurs prédéfinies. Pour déclarer une énumération on doit utiliser le mot clé **enum** en spécifiant le nom de cette énumération, par exemple

```
0 références
public enum EnumExample
{
}
```

Le quatrième genre de type est le délégué. C'est une sorte de pointeur vers une fonction. Pour déclarer un délégué on doit utiliser le mot clé **delegate** en spécifiant le nom de ce délégué, par exemple

```
public delegate int DelExample(string str);
```

Ici le délégué n'a pas de corps et il doit représenter la signature des fonctions (ou méthodes) qui peut les pointer. Dans notre exemple la fonction doit retourner un entier et accepte un seul argument de type chaîne de caractères.

L'interface est un autre genre pour un type. Elle permet de déclarer des contrats. C'est en quelque sorte une classe sans code (ou implémentation). Les interfaces sont idéales pour faire des abstractions et rendre le code très robuste. Pour déclarer une interface on doit utiliser le mot clé **interface** en spécifiant le nom de cette interface, par exemple

0 références

```
public interface INterExample
{
}
```

Membres d'un type

On a vu précédemment les différents genres d'un type, dans cette partie on va voir ce que peut contenir un type. Après la déclaration d'un type on déclare ce qu'on appelle les membres.

Le premier genre d'un membre est le champ. Un champ est le seul membre qui lui est associé une zone mémoire. Plus on a de membres plus le type occupe davantage de mémoire. Voici un exemple de déclaration d'un champ

```
public class ClassExample
{
    int field = 1;
}
```

Ici on a déclaré le champ **field** de type **int** avec **1** comme valeur initiale. La taille du champ est celle du type **System.Int32** donc 32 bits soit 4 octets.

Le deuxième genre d'un membre est la méthode. En programmation orienté objet, méthode est le nom donné pour la fonction. Voici un exemple de déclaration d'une méthode

```
public class ClassExample
{
    0 références
    int MaMethode(int p)
    {
        return 2 * p;
    }
}
```

Ici on a déclaré la méthode **MaMethode** avec comme paramètre **p** de type **int**. Elle retourne une valeur de type **int** aussi.

Les types qui sont concernés par les méthodes sont les classes, les structures et les interfaces.

Le troisième genre d'un membre est le constructeur. C'est une méthode spéciale qui ne retourne rien et dont le nom doit être celui de son type. Le rôle principal d'un constructeur est l'initialisation d'une instance d'un type. Voici un exemple

2 références

```
public class ClassExample
{
    int field;
    0 références
    ClassExample(int f)
    {
        field = f;
    }
}
```

Ici on a déclaré le constructeur qui accepte comme paramètre un entier.

Les types qui sont concernés par les constructeurs sont les classes et les structures.

Si un type n'est pas déclaré alors un constructeur par défaut sera automatiquement ajouté. Le constructeur par défaut n'accepte aucun argument et ne contient aucun code. Il est possible de déclarer explicitement le constructeur par défaut dans le cas d'une classe mais pas dans le cas d'une structure.

Le quatrième genre d'un membre est la propriété. C'est une nouveauté comparée aux autres langages orientés objet comme Java. La propriété permet de limiter l'accès aux données d'un type en l'obligeant de passer par deux méthodes spéciales appelées **getter** et **setter**. La propriété est utilisée comme un champ que ce soit lors de l'affectation ou la lecture. Voici un exemple de déclaration d'une propriété

```
public class ClassExample
{
    int field = 1;
    0 références
    int MaPropriete
    {
        get { return 2 * field; }
        set { field = value / 2; }
    }
}
```

Ici on a déclaré la propriété **MaPropriete** de type **int** et qui contient un getter et un setter. Il est possible d'omettre le setter ou le getter (mais pas les deux au même temps). Dans le corps du setter il est possible d'utiliser le mot clé **value** qui est le nom donné à la valeur d'une affectation. Voici un exemple d'utilisation

```
int MaMethode(int p)
{
    MaPropriete = p;
    return 3 * MaPropriete;
}
```

En général les propriétés sont utilisées pour interdire l'affectation et n'autoriser que la lecture, c'est à dire n'utiliser que le getter. Les bonnes pratiques de codage obligent de ne pas exposer directement les champs d'un type (que ce soit en lecture ou écriture) et d'utiliser les propriétés à la place. Dans certains usages on est obligé de faire la déclaration d'un champ et de la propriété associée. Ceci rend le code

relativement lourd. Il est possible de faire plus simple en utilisant les propriétés auto-implémentées. Voici un exemple

```
public class ClassExample
{
    0 références
    int MaProprieteAuto
    {
        get;
        set;
    }
}
```

Ici on a déclaré la propriété auto-implémentée **MaProprieteAuto**. Le champ associé à la propriété est implicitement déclaré. Le getter retourne sa valeur et le setter la modifie. Il est possible d'initialiser la valeur d'une propriété auto-implémentée comme suit

```
public class ClassExample
{
    0 références
    int MaProprieteAuto
    {
        get;
        set;
    } = 1;
}
```

Pour clôturer cette partie notez qu'un membre peut être déclaré statique. Les membres statiques s'appellent membres de type et se déclare via le mot clé **static**. Les membres non statiques s'appellent les membres d'instance. L'utilisation d'un membre statique se fait directement à partir du nom du type. Exemple de déclaration

```
public class ClassExample
{
    0 références
    public static void MaMethodeStatique()
    {
    }
}
```

et voici son utilisation

```
static void Main(string[] args)
{
    ClassExample.MaMethodeStatique();
}
```

Pour le type énumération, ces membres sont déclarés d'une façon spéciale. Juste on déclare la liste des valeurs. Voici un exemple


```
public enum EnumExample
{
    Lundi,
    Mardi,
    Mercredi
}
```

Il est possible d'affecter explicitement la valeur d'une énumération, dans ce cas les valeurs qui suivent sont une incrémentation de cette valeur par exemple

```
public enum EnumExample
{
    Lundi,
    Mardi = 5,
    Mercredi
}
```

Dans cet exemple **Mercredi** prendra la valeur **6**.

Encapsulation

Avec la programmation orientée objet il est possible de restreindre l'accès aux types et aux membres des types. On appelle ceci l'encapsulation. Chaque type ou membre a une certaine visibilité. La visibilité est fixée via les modificateurs d'accès et il y en a quatre dont les mots clés sont **private**, **protected**, **internal** et **public**.

La visibilité la moins restrictive est la privée fixée via le mot clé **private**. Les membres privés d'un type ne sont visibles qu'à l'intérieur de ce type.

Par la suite on trouvera la visibilité protégée fixée via le mot clé **protected**. Les membres protégés sont visibles à l'intérieur du type et les types qui y héritent (on verra l'héritage dans la suite).

La visibilité interne fixée par le mot clé **internal** permet aux membres d'être visibles seulement dans l'assembly (dll ou projet).

Enfin la visibilité la plus étendue est la publique fixée par le mot clé **public**. Un membre public est visible partout.

Notez qu'il est possible d'avoir la combinaison entre la visibilité interne et protégée via le mot clé **internal protected** ou privée et protégée via le mot clé **private protected**. Voici des exemples de déclarations

```

public class ClassExample
{
    private int field;
    0 références
    protected ClassExample()
    {}
    0 références
    internal void M() { }
    0 références
    public int Property { get; set; }
    0 références
    internal protected void M2() { }
    0 références
    private protected void M3() { }
}

```

Les types n'acceptent que les visibilités publiques et internes.

Dans le cas d'une propriété il est possible de restreindre la visibilité du getter ou du setter. Voici un exemple

```

public int Property { get; private set; }

```

La déclaration de la visibilité (d'un type ou membre) est facultative, si elle n'est pas spécifiée alors elle sera celle qui donne la moindre visibilité. Dans ce cas les types seront internes et les membres privés.

Types génériques

Lors de la déclaration d'un champ, ou une variable ou les paramètres d'une méthode on a besoin de spécifier le type et il sera impossible de le remplacer par un autre sans modifier une partie du code. Il est possible en C# de spécifier un type dont on ne connaît pas l'implémentation à l'avance. Ce sont les types génériques. Pour spécifier des types génériques il suffit de mettre au niveau du type une liste de noms après la déclaration du type en question en utilisant les symboles < et > (<> est appelé diamant). Voici un exemple avec une classe

```

class GenericExample<T1,T2>
{
}

```

et avec un autre avec un délégué

```

delegate void DelExample<T>();

```

Une fois les paramètres génériques déclarés il est possible de l'utiliser dans le type. Par exemple

```

class GenericExample<T1,T2>
{
    T1 field;
    0 références
    GenericExample(T1 fValue)
    {
        field = fValue;
    }
    0 références
    T2 Method()
    {
        return default(T2);
    }
}

```

Il est possible de restreindre les types d'un paramètre générique à certains genres de type ou des sous classes. Par exemple

```

class GenericExample<T1,T2> where T1 : Exception where T2 : struct

```

Si un champ ou une variable n'est pas initialisée alors il prendra sa valeur par défaut. La valeur par défaut des types qui ne sont pas des structures est **null**. La valeur par défaut d'une structure varie d'une structure à une autre. Par exemple la valeur par défaut de **int** est **0** et la valeur par défaut de **DateTime** est **1/1/1**. Le mot clé **default** retourne la valeur par défaut d'un type (exemple **default(int)**).

Il est possible de déclarer un type associé à une structure en utilisant le mot clé **?**. Ce sont les types nullable. Contrairement aux structures, les types nullable peuvent avoir une valeur **null**. Voici un exemple

```

public class ClassExample
{
    0 références
    int Method ()
    {
        // int e = null;
        int? e = null;
        e = 1;

        return e.Value;
    }
}

```

La propriété **Value** d'un type nullable retourne sa valeur (dans notre exemple c'est 1).

Collections

En .NET, il existe un ensemble de classes qui facilitent la manipulation des collections.

Le premier genre de collection est le tableau. Pour déclarer un tableau on doit utiliser les crochets ouvrant et ferment **[]** sans oublier le type. Par exemple

```
int Method ()
{
    int[] t = new int[5];

    return t[0];
}
```

Ici on a déclaré un tableau de 5 éléments dont le contenu de chaque élément est la valeur par défaut de **int**. Par accéder à la valeur d'un élément on utilise à nouveau les crochets en spécifiant l'indice (les indices commencent à partir de 0). Il est possible d'initialiser les valeurs d'un tableau par exemple

```
int Method ()
{
    int[] t = new int[] {-5, 1, 0, 6};

    return t[0];
}
```

La taille du tableau est automatiquement calculée donc on n'a pas besoin de la spécifier. La propriété **Length** donne la taille d'un tableau. Tous les tableaux héritent de **Array**.

Une fois un tableau est créé il est impossible de changer sa taille. Pour travailler avec des listes dynamiques il y a les listes génériques. La classe associée se nomme **List** (espace de noms **System.Collections.Generic**). L'ajout d'un élément se fait via la méthode **Add** et l'accès aux éléments via l'indexeur de cette classe (propriété spéciale nommée **this**). Il est possible de peupler une liste en utilisant une syntaxe proche de l'initialisation d'un tableau. Voici un exemple résumant

```
int Method ()
{
    List<int> l = new List<int>() { -1, 5, 0, 7};

    l.Add(5);
    return l[0];
}
```

La propriété **Count** retourne la taille de la liste. Ici la taille de la liste est 5.

Si on a une liste de valeurs, alors il est possible de les parcourir valeur par valeur via le mot clé **foreach**, par exemple

```

void Methode()
{
    int[] liste = new int[] { -2, 1, 5, 6 };
    int sum = 0;

    foreach (var item in liste)
    {
        sum += item;
    }
}

```

Le mot clé **foreach** ne s'applique qu'aux types qui implémentent l'interface **IEnumerable** (voir partie héritage).

Héritage

L'héritage est l'une des particularités importantes de l'orienté objet. Avec l'héritage il est possible d'étendre un type. L'héritage concerne les genres classe et interface. Si un type hérite d'un autre alors tous les membres deviennent la propriété de ce nouveau type. Cependant la visibilité restera en vigueur et certains membres resteront invisibles (comme les membres privés et internes). La taille occupée par une classe fille (la classe qui a hérité) est la somme de sa propre taille et la taille occupée par sa classe mère.

Pour déclarer un héritage on utilise le symbole « : » suivi par le type mère. Voici un exemple d'un héritage de classe

```

class ClasseFille : ClassExample
{
}

```

On .NET il est impossible de faire un héritage de plusieurs classes, seule un héritage simple est possible. Cependant il est possible de faire un ou plusieurs héritages à partir d'une ou plusieurs interfaces. Le terme approprié pour ce mécanisme est l'implémentation. Ainsi une classe peut hériter d'une autre classe et peut implémenter plusieurs interfaces. Lors de l'écriture d'un héritage il faut mettre la classe au début. Dans le cas où la classe mère n'est pas spécifiée elle sera la classe **System.Object**. Cette classe est la classe mère de toutes les classes et structures. Voici un exemple d'héritage multiple

```

class ClasseFille : ClassExample, INterExample, ICloneable
{
    0 références
    public object Clone()
    {
        throw new NotImplementedException();
    }
    1 référence
    public void M()
    {
        throw new NotImplementedException();
    }
}

```

Ici les deux méthodes **Clone** et **M** sont deux méthodes de la classe **ClasseFille** et d'une façon implicite les implémentations des interfaces **INterExample** et **ICloneable**. Pour faire une implémentation implicite il faut que le membre soit public. Il est possible de faire une implémentation explicite en spécifiant l'interface associée. Exemple

```

class ClasseFille : INterExample
{
    1 référence
    void INterExample.M()
    {
        throw new NotImplementedException();
    }
}

```

Une interface peut hériter d'une autre interface dans ce cas si une classe ou une structure implémente l'interface fille alors elle doit implémenter les membres de l'interface fille. Un exemple d'héritage d'interface

```

public interface INterExample : ICloneable
{
    0 références
    void M();
}

```

Il est possible d'interdire l'héritage à partir d'une classe donnée. Dans ce cas il sera impossible de prendre cette classe comme étant une classe mère d'une autre. Le mot clé à utiliser est **sealed** et voici un exemple

```

sealed class ClassExample
{
}

```

Les structures sont des types spéciaux. En effet une structure hérite d'une classe spéciale appelée **ValueType**. Cet héritage est implicite. Une autre particularité des structures est qu'elles sont **sealed**. Donc il est impossible d'hériter d'une structure.

Polymorphisme

En orienté objet le polymorphisme est la capacité de changer de comportement. En .NET ce changement de comportement se fait via les surcharges. Le premier type de surcharge est la surcharge Ad-hoc en déclarant plusieurs constructeurs ou plusieurs méthodes du même nom. La différence doit se faire au niveau de l'ordre et les types des paramètres. Voici un exemple

```
class ClassExample
{
    0 références
    ClassExample()
    {}
    0 références
    ClassExample(int v)
    {}
    0 références
    void M() {}
    0 références
    int M(int p)
    {
        return 2 * p;
    }
    0 références
    void M(string p)
    {}
}
```

Le type de retour n'est pas pris en compte pour différencier les méthodes surchargées.

Il est possible de déclarer des méthodes (ou propriétés) dont le comportement peut varier selon l'instance utilisée.

Le premier comportement est assuré via les méthodes (ou propriétés) virtuelles. Pour déclarer une méthode virtuelle on utilise le mot clé **virtual**. La substitution dans une classe fille est déclarée via le mot clé **override**. Voici un exemple

```
2 références
public class ClassExample
{
    3 références
    public virtual void M()
    {
        Console.WriteLine("ClassExample.M");
    }
}
2 références
public class ClasseFille : ClassExample
{
    3 références
    public override void M()
    {
        Console.WriteLine("ClasseFille.M");
    }
}
```

Le code exécuté dépend de l'instance utilisée. Voici un exemple

```
static void Main(string[] args)
{
    ClassExample c = new ClassExample();

    c.M(); // Execute ClassExample.M
    c = new ClasseFille();
    c.M(); // Execute ClasseFille.M
}
```

Il est possible de déclarer une méthode (ou propriété) qui aura le même comportement mais sans corps. Ce sont les méthodes abstraites. Le mot clé utilisé est **abstract**. La classe associée doit être aussi déclarée abstraite. Il est impossible de créer une nouvelle instance d'une classe abstraite (car le code est incomplet). Comme avec les méthodes abstraites la substitution se fait via le mot clé **override**. Voici un exemple de déclaration

```
3 références
public abstract class ClassExample
{
    4 références
    public abstract void M();
}
1 référence
public class ClasseFille1 : ClassExample
{
    4 références
    public override void M()
    {
        Console.WriteLine("ClasseFille1.M");
    }
}
0 références
public class ClasseFille2 : ClassExample
{
    4 références
    public override void M()
    {
        Console.WriteLine("ClasseFille2.M");
    }
}
```

et un exemple d'utilisation

```
static void Main(string[] args)
{
    ClassExample c;

    c = new ClasseFille1();
    c.M(); // Execute ClasseFille1.M
    c = new ClasseFille2();
    c.M(); // Execute ClasseFille2.M
}
```


Parfois on a besoin d'appeler la méthode de base dans ce cas on peut utiliser le mot clé **base**. On peut aussi appeler les constructeurs de la classe de base. Le mot clé **this** permet de pointer vers les membres du type. Voici un exemple

```

6 références
public class ClassExample
{
    2 références
    protected ClassExample(int f){}
    0 références
    ClassExample() : this(0)
    {}
    4 références
    public virtual void M()
    {Console.WriteLine("ClassExample.M");}
}
1 référence
public class ClasseFille : ClassExample
{
    0 références
    ClasseFille() : base(1)
    {}
    4 références
    public override void M()
    {
        Console.WriteLine("ClasseFille.M");
        base.M();
    }
}

```

Instanciation

Une fois un type est déclaré, il est possible de faire une instance de lui. Le mot clé utilisé est **new** suivi par le constructeur. Il est impossible de faire une instance d'une interface ou d'une classe abstraite. Voici des exemples

```

static void Main(string[] args)
{
    ClassExample c = new ClassExample(5);
    StructExample s = new StructExample();
    int i = new int();
}

```

De la même façon, pour un délégué on peut utiliser le mot clé **new**, mais ce n'est pas obligatoire. Voici des exemples

```

delegate void DelExample1(int p);
delegate int DelExample2();
5 références
public class ClassExample
{
    1 référence
    static void M1(int p)
    {}
    1 référence
    int M2()
    { return 0; }
    0 références
    static void ExecuteDel()
    {
        DelExample1 del1 = new DelExample1(M1);
        del1(5);

        DelExample2 del2 = new ClassExample().M2;
        del2();
    }
}

```

Si on veut faire une instance d'un type générique on doit spécifier le ou les types au niveau du constructeur. Par exemple

```

public class ClassExample<T>
{
    0 références
    static ClassExample<int> GetInstanceWithInt()
    {
        return new ClassExample<int>();
    }
    0 références
    static ClassExample<T> GetInstance()
    {
        return new ClassExample<T>();
    }
}

```

Il est possible de faire une instanciation et une initialisation dans la même ligne (initialisation *inline*). Par exemple

```

public class ClassExample
{
    1 référence
    public int P1 { get; set; }
    1 référence
    public string P2 { get; set; }
    0 références
    static ClassExample GetInstance()
    {
        return new ClassExample() { P1 = 5, P2 = "V" };
    }
}

```

Dans le cas d'une liste, les éléments peuvent être ajoutés en utilisant la même technique. Exemple

```
static void Main(string[] args)
{
    List<int> l = new List<int> {-5, 5, 6, 8 };
}
```

Ici on remarque que les parenthèses du constructeur ont été omises. Ceci est possible dans le cas d'une initialisation en ligne avec un constructeur par défaut. Ce type de pratique s'appelle simplification et c'est parmi les bonnes pratiques.

Il y a une autre manière de faire des instances en utilisant des types anonymes. Voici un exemple

```
static void Main(string[] args)
{
    var t = new { P1 = 2, P2 = "v" };
}
```

Ici il y a création d'une instance d'un type spécial qui ne peut pas être utilisé ailleurs. Le type est inconnu lors du codage, c'est pourquoi il est impossible de l'utiliser en dehors de sa portée. L'utilisation du mot clé **var** est obligatoire. Le type créé est une classe **sealed** qui hérite de **object**. Elle contient deux propriétés de type **int** et **string** initialisées avec les valeurs 2 et « v ».

Passage d'un paramètre

Lors de l'appel d'une méthode (ou constructeur) la valeur d'un paramètre peut être une référence (sorte de pointeur) vers une instance ou bien une copie. Si on passe une copie alors la valeur d'origine ne sera pas modifiée (puisque l'on a passé une copie). Si on passe une référence alors toute modification affectera la valeur d'origine (car c'est un pointeur). Par défaut toute instance d'une classe est passée par référence et toute instance d'une structure ou énumération est passée par valeur. Les types nullable sont des structures donc leurs instances seront passées par valeur. Exemple

```

public class ClassExample
{
    3 références
    struct StructExample
    {
        3 références
        public int P { get; set; }
    }
    3 références
    int P { get; set; }
    1 référence
    static void Modify(ClassExample c, StructExample s)
    {
        c.P = -2;
        s.P = -2;
    }
    0 références
    static void Test()
    {
        ClassExample c = new ClassExample { P = 2 };
        StructExample s = new StructExample { P = 2 };
        Modify(c, s);
        Console.WriteLine(c.P); // Affiche -2
        Console.WriteLine(s.P); // Affiche 2
    }
}

```

Notez dans cet exemple l'utilisation des types imbriqués (nested types) via la structure **StructExample**.

Inférence de type

Lors de la déclaration d'une variable locale (dans une méthode ou constructeur) il est possible de ne pas déclarer son type si le compilateur peut en déduire du contexte de codage. Ce comportement s'appelle inférence de type. Le mot clé **var** doit être utilisé pour faire usage de l'inférence de type. Voici un exemple

```

static void Main(string[] args)
{
    var str = "1780";
    var e = 15;
    var l = 15L;

    int length = str.Length;
    int c = e.CompareTo(15);
    var t = e > l;
}

```

Le fait d'utiliser **var** ne veut pas dire que le type est inconnu. En effet dans l'exemple ci-dessus le compilateur a identifié la variable **str** comme étant une chaîne de caractères car on voit par la suite l'utilisation de la propriété **Length**.

Type DateTime

Le type **DateTime** est une structure de l'espace de noms **System** dont voici une description de certains membres :

Le constructeur **DateTime(int year, int month, int day)** : construit une nouvelle instance d'une date.

La propriété **DateTime.Now** : retourne la date en cours.

Les propriétés **DateTime.Year**, **DateTime.Day**, **DateTime.Hour** : retourne respectivement l'année, le jour et l'heure d'une date.

Opérateur + : retourne une date à laquelle on a ajouté une durée (Type **TimeSpan**).

La structure **TimeSpan** contient des propriétés qui permettent de convertir la durée en un total par exemple un total de jours. Parmi ces propriétés retournant des totaux on a **TimeSpan.TotalDays**, **TimeSpan.TotalHours**, **TimeSpan.TotalMinutes** ou encore **TimeSpan.TotalSeconds**.

Opérateur - : retourne une durée résultat de la différence entre deux dates ou résultat de la différence entre une date et une durée.

Opérateurs ==, !=, >, >=, <, <= : comparent deux dates ou deux durées.

Exemples d'utilisation de ces deux types :

```
DateTime now = DateTime.Now;
DateTime birthDay = new DateTime(1980, 3, 15);
TimeSpan age = now - birthDay;

Console.WriteLine(age.TotalDays);
DateTime dateAge60 = birthDay + new TimeSpan(365 * 60, 0, 0, 0);

Console.WriteLine(dateAge60.Year);

DateTime birthDay2 = new DateTime(1950, 2, 4);

Console.WriteLine(birthDay2 > birthDay);
```

Le résultat d'affichage est le suivant

```
15420,71970886701
2040
False
```

Méthodes d'extensions

Les méthodes d'extensions permettent d'étendre les fonctionnalités d'une classe. Ces méthodes ne sont pas réellement ajoutées au type (classe ou structure) mais elles seront utilisées comme si elles font partie. Les méthodes d'extensions ne seront activées que sur les instances. La déclaration d'une méthode d'extension se fait dans une classe statique. Une classe statique est une classe **sealed** dont tous les membres doivent être statiques. Voici un exemple de classe statique

```
public static class Helper
{
    public static readonly float Pi = 3.14f;
    1 référence
    static float DoublePi { get { return 2 * Pi; } }
    0 références
    public static float CirclePerimeter(float r)
    {
        return DoublePi * r;
    }
}
```

Pour ajouter une extension, on doit choisir le type, par exemple on va étendre le type **float** en rendant la méthode **CirclePerimeter** une extension. Dans ce cas il suffit d'ajouter le mot clé **this** comme suit

```
]namespace LibExample
{
    0 références
    public static class Helper
    {
        public static readonly float Pi = 3.14f;
        1 référence
        static float DoublePi { get { return 2 * Pi; } }
        0 références
        public static float CirclePerimeter(this float r)
        {
            return DoublePi * r;
        }
    }
}
```

Une fois l'extension déclarée, il suffit par la suite d'inclure l'espace de noms en utilisant un **using** et l'extension apparaît. Par exemple

```
static void Main(string[] args)
{
    float r = 4.5f;
    Console.WriteLine($"Périmètre : {r.CirclePerimeter()}");
}
```

Fonctions anonymes

Il est possible de déclarer une fonction qui ne fait pas partie d'une classe mais dans une méthode ou un constructeur de cette classe. Ce sont les fonctions anonymes. Pour déclarer une fonction anonyme on a

besoin d'un prototype. Ceci est assuré via les délégués. La déclaration se fait en utilisant le mot clé **delegate**. Voici un exemple

```
public delegate int MultiplyBy(int by);  
0 références  
public static class Helper  
{  
    0 références  
    public static void MultiplyBy2And3(int v)  
    {  
        MultiplyBy m = delegate (int by)  
        {  
            return v * by;  
        };  
  
        Console.WriteLine(m(2));  
        Console.WriteLine(m(3));  
    }  
}
```

On voit ici que la fonction anonyme peut accéder aux paramètres de la méthode. Elle peut aussi accéder aux variables locales. Il existe deux délégués prédéfinis qui peuvent être utilisés : **Func** et **Action**. Les deux délégués sont génériques permettant de spécifier les types des paramètres. Le délégué **Func** permet de pointer vers une méthode ou une fonction avec une valeur de retour alors que **Action** sans valeur de retour. Dans le cas de **Func** le dernier type est celui du retour. Voici une des déclarations de **Func**

```
...public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

et une des déclarations de **Action**

```
...public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
```

Expressions lambda

Il est possible de rendre l'écriture des fonctions anonymes encore plus simple en utilisant une syntaxe appelée expression lambda. Si on reprend l'exemple précédent, l'écriture de la fonction anonyme devient

```
MultiplyBy m = (int by) =>  
{  
    return v * by;  
};
```

Ici on a remplacé le mot clé **delegate** par le symbole **=>**. On peut encore simplifier en supprimant le type du paramètre, la parenthèse et le mot clé **return**. La fonction anonyme devient comme ceci

```
MultiplyBy m = by => v * by;
```

LINQ

Le **LINQ** (Language **IN**tegrated **Q**uery) est un sous langage de C# qui permet d'unifier l'interrogation des sources de données. Les mots clés utilisés dans ce sous langage sont intégrés dans la syntaxe de C#. Une source de données peut être un tableau ou une liste en mémoire, ou une base de données ou un document XML ou bien un autre type de source. Les sources implémentées par défaut avec le .NET sont les objets, les documents XML, les bases de données compatibles avec le langage SQL, les entités et les DataSets. Lorsqu'on forme une requête l'exécution peut être différée. En général l'exécution se fait lors de la première utilisation du résultat.

Méthodes d'extensions du LINQ

Le **LINQ** est principalement des méthodes d'extensions qui se trouvent dans la classe **System.Linq.Enumerable**. On doit donc ajouter l'espace de nom **System.Linq** dans les clauses **using**.

Ces méthodes s'appliquent à tout énumérable fortement typé (type implémentant l'interface **IEnumerable<T>**).

La première méthode d'extension du **LINQ** est le **Select**. Elle permet de modifier le type des éléments d'une source. Par exemple

```
static void Main(string[] args)
{
    var t = new[] { -1, 2, 3, 5 };

    t.Select(e => e.ToString());
}
```

Ici on a modifié la liste des entiers du tableau en une liste énumérable de chaîne de caractères. La liste retournée est de type **IEnumerable<string>**. On remarque l'utilisation d'une expression lambda.

Les méthodes **First**, **Last** et **Single** permettent de retourner un seul élément. La méthode **Single** retourne le seul élément d'une liste. Si la liste contient plus qu'un élément alors une exception est générée. Par exemple


```
static void Main(string[] args)
{
    var t1 = new[] { -1, 2, 3, 5 };
    var t2 = new[] { 7 };

    t1.First(); // Retourne -1
    t1.Last(); // Retourne 5
    t1.Single(); // Lève une exception
    t2.Single(); // Retourne 7
}
```

Si la liste est vide alors les trois méthodes lèvent une exception. Pour éviter cette exception on peut utiliser les méthodes **FirstOrDefault**, **LastOrDefault**, **SingleOrDefault**. Ils retournent la valeur par défaut de la liste si elle est vide (0 dans notre exemple).

Il est possible de faire un filtrage de la liste en sélectionnant quelques éléments. Le filtrage peut se faire via les méthodes **Take**, **Skip** et **Where**. Exemple

```
static void Main(string[] args)
{
    var t = new[] { -1, 2, 3, 5 };

    t.Take(2); // Retourne {-1, 2}
    t.Skip(2); // Retourne {3, 5}
    t.Where(e => e > 0); // Retourne {2, 3, 5}
    t.Where(e => e > 0 && e < 4); // Retourne {2, 3}
}
```

Il est possible de faire usage du **Fluent Interface** en combinant plusieurs méthodes via un mécanisme de chaînage. Par exemple

```
static void Main(string[] args)
{
    var t = new[] { -1, 2, 3, 5 };

    t.Where(e => e > 0).
      Select(e => 2 * e).
      Select(e => "c" + e.ToString()); // Retourne {"c4", "c6", "c10"}
}
```

Il est possible de trier une liste via les méthodes **OrderBy** et **OrderByDescending**. Exemple

```
static void Main(string[] args)
{
    var t = new[] { 5, 3, -1, 2 };

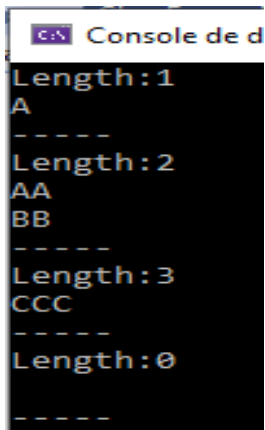
    t.OrderBy(e => e); // Retourne {-1, 2, 3, 5}
    t.OrderByDescending(e => e); // Retourne {5, 3, 2, -1}
    t.OrderBy(e => -e); // Retourne {5, 3, 2, -1}
}
```

Il est possible de grouper la liste via la méthode **GroupBy**. Par exemple

```
static void Main(string[] args)
{
    var t = new[] { "A", "AA", "BB", "CCC", "" };
    IEnumerable<IGrouping<int, string>> r = t.GroupBy(e => e.Length);

    foreach (var group in r)
    {
        Console.WriteLine($"Length:{group.Key}");
        foreach (var item in group)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("-----");
    }
}
```

Le résultat est le suivant



```

c:\> Console de d
Length:1
A
-----
Length:2
AA
BB
-----
Length:3
CCC
-----
Length:0
-----

```

Enfin il existe des méthodes pour faire des opérations sur les éléments (méthodes d'agrégations), comme **Min**, **Max**, **Sum**, **Count**...Voici un exemple

```
static void Main(string[] args)
{
    var t = new[] { 2, -1, 5, 3 };

    t.Min(); // Retourne -1
    t.Max(); // Retourne 5
    t.Count(); // Retourne 4
    t.Sum(); // Retourne 9
}
```

LINQ intégré

Ici on va voir comment faire usage des mots clé du langage pour faire des requêtes **LINQ**. Pour une bonne partie des méthodes d'extensions ils existent des équivalents en utilisant des requêtes via le **LINQ** intégré.

Pour interroger la source avec le **LINQ** intégré, on doit commencer par le mot clé **from** et finir par le mot clé **select** en spécifiant la source via le mot clé **in**. Par exemple

```
static void Main(string[] args)
{
    var t = new[] { 2, -1, 5, 3 };

    var r = from e in t select "c" + e.ToString();
    var l = r.ToList(); // Retourne {"c2", "c-1", "c5", "c3"}
}
```

Contrairement aux méthodes d'extensions qui sont instantanées les requêtes via le **LINQ** intégré sont différées. C'est pourquoi dans l'exemple ci-dessus on a utilisé **ToList** pour avoir un résultat. Parcourir une requête l'oblige à être exécutée.

Pour faire un filtrage on utilise le mot clé **where**. Exemple

```
static void Main(string[] args)
{
    var t = new[] { 2, -1, 5, 3 };
    var r = from e in t where e > 0 select e;

    foreach (var item in r)
    {
        Console.WriteLine(item); // Affiche 2 puis 5 puis 3
    }
}
```

Le tri ascendant se fait via le mot clé **orderby** et le tri descendant en ajoutant le mot clé **descending**. Exemple

```
static void Main(string[] args)
{
    var t = new[] { 2, -1, 5, 3 };
    var r1 = from e in t orderby e select e; // Retourne {-1, 2, 3, 5}
    var r2 = from e in t orderby e descending select e; // Retourne {5, 3, 2, -1}
}
```

Le groupement se fait en utilisant les mots clés **group** et **by**. Exemple

```
static void Main(string[] args)
{
    var t = new[] { 2, -1, 2, -1, 2, 3 };
    IEnumerable<IGrouping<int,int>> r = from e in t group e by e;
}
```

Il est possible de stocker le résultat d'une requête dans une variable locale à l'intérieur de la requête, ce qui permet de l'utiliser dans la suite. Par exemple

```
static void Main(string[] args)
{
    var t = new[] { -1, 2, 3, 5 };
    var r = from e in t select 2 * e into e2 where e2 % 2 == 0 select e2 / 2;
}
```

Ici on récupère toute la liste.

Il est possible d'interroger plusieurs sources au même temps. Par exemple

```
static void Main(string[] args)
{
    var t1 = new[] { -1, 2, 3, 5 };
    var t2 = new[] { 2, 5 };
    var r = from e1 in t1 from e2 in t2 where e1 == e2 select e1 * e2; // Résultat {4, 25}
}
```

Compléments

Exceptions

Une des nouveautés de certains langages orientés objet est la gestion plus efficace des exceptions (les erreurs). Pour provoquer ou générer une exception on utilise le mot clé **throw**. La génération ne

concerne que les classes qui héritent de la classe **Exception**. Voici un exemple de déclaration d'une exception

```
class ExceptionExample : Exception
{
}
```

et un exemple de génération

```
void Methode()
{
    throw new ExceptionExample();
}
```

Pour traiter une exception on doit la capter. La capture se fait dans un bloc **try...catch...finally**.

Le code qui risque de provoquer l'exception doit être dans le bloc **try**. Le traitement dans le bloc **catch** et enfin le bloc **finally** pour faire un traitement dans tous les cas (exception provoquée ou non). Voici un exemple

```
void MethodeAvecException()
{
    throw new ExceptionExample();
}
0 références
void Method ()
{
    try
    {
        Console.WriteLine("Ligne exécutée - 1");
        MethodeAvecException();
        Console.WriteLine("Ligne non exécutée");
    }
    catch(ExceptionExample exception)
    {
        Console.WriteLine(exception);
    }
    finally
    {
        Console.WriteLine("Ligne exécutée - 2");
    }
}
```

Notez qu'il est possible de filtrer les exceptions à traiter selon d'autres critères, par exemple le contenu de la propriété **Message**. Le mot clé à utiliser est **when**. Voici un exemple

```
try
{
    // Code
}
catch(SmtpException exception) when (exception.StatusCode == SmtpStatusCode.MailboxUnavailable)
{ }
```

Conversion de type

En .NET il est possible de convertir une instance d'un type donnée en une instance d'un autre type. Cette conversion peut se faire d'une façon implicite dans le cas où il n'y a pas de perte d'informations sinon il faut une conversion explicite via le casting ou via des méthodes dédiées. Voici des exemples

```
static void Main(string[] args)
{
    int e = 15;
    long l = e; // Conversion implicite
    short s = (short)e; // Conversion explicite via un cast
    string str = Convert.ToString(e); // Conversion explicite
                                     // via la méthode Convert.ToString
}
```

Dans le cas où un cast est impossible (détecté lors de l'exécution) une exception de type **InvalidCastException** est levée. Si on veut éviter cette exception on peut faire un test en utilisant les deux mots clés **is** et **as**. Le mot clé **is** teste si une instance peut être convertie en un type cible sans perte et le mot clé **as** fait le même test mais en affectant la valeur **null** si la conversion est impossible. Voici des exemples

```
static void Main(string[] args)
{
    object e = 15;
    bool testIs = e is int;
    ICloneable testAs = e as ICloneable;

    if (testAs != null)
    {
        int eClone = (int)testAs.Clone();
    }
}
```

Ici la structure **int** n'implémente pas l'interface **ICloneable**.

Il existe une classe dédiée à la conversion, c'est la classe **Convert**. Elle contient plusieurs méthodes permettant de convertir n'importe quel type vers un autre (dans la limite du possible). On peut fixer les manières de convertir un type donné (vers d'autre) en implémentant l'interface **IConvertible**.

Certains types contiennent la méthode **Parse**. Cette méthode permet de convertir vers ce type en utilisant d'avantages de paramètres par exemple on peut fixer le format de conversion. La méthode **ToString** est une méthode qui peut être utile pour la conversion vers une chaîne de caractères. La méthode **ToString** sans paramètre est présente pour n'importe quel type car c'est une méthode de classe **objet**.

Lors de l'utilisation d'une méthode de conversion, il est possible de fixer la culture de conversion (culture française, culture anglaise, culture invariante...). La classe pour les cultures se nomme **System.Globalization.CultureInfo**. Voici des exemples de conversions

```
static void Main(string[] args)
{
    const string str = "1780";
    int e;
    e.. = Convert.ToInt32(str);
    e.. = (int)Convert.ChangeType(str, typeof(int));
    e.. = Convert.ToInt32(str, CultureInfo.InvariantCulture);
    e = int.Parse(str, NumberStyles.Integer);
    string s.. = e.ToString("### ###"); // Affiche 1 780
}
```

Complément types

L'environnement .NET fournit plusieurs types prédéfinis de plusieurs genres (des classes, des structures, des délégués, des interfaces). Parmi ces types prédéfinis il y a les types primitifs dont voici une liste

Type	Classe	Valeur par défaut	Valeur minimum	Valeur maximum
sbyte	System.SByte	0	-128	127
byte	System.Byte	0	0	255
short	System.Int16	0	-32768	32767
ushort	System.UInt16	0	0	65535
int	System.Int32	0	-2147483648	2147483647
uint	System.UInt32	0u	0	4294967295
long	System.Int64	0L	-9223372036854775808	9223372036854775807
ulong	System.UInt64	0u	0	18446744073709551615
float	System.Single	0.0f	$\pm 1.5 \times 10^{-45}$	$\pm 3.4 \times 10^{38}$
double	System.Double	0.0d	$\pm 5.0 \times 10^{-324}$	$\pm 1.7 \times 10^{308}$
decimal	System.Decimal	0.0m	$\pm 1.0 \times 10^{-28}$	$\pm 7.9 \times 10^{28}$
bool	System.Boolean	false	Deux valeurs possibles : <i>true</i> et <i>false</i>	
char	System.Char	'\u0000'	'\u0000'	'\uffff'
object	System.Object	null	-	-
string	System.String	null	-	-

Les types de la liste ci-dessus sont des structures sauf pour les deux derniers qui sont des classes.

Les noms courts comme **int** ou **long** sont appelés alias. Le type réel est celui de la deuxième colonne.

Pour clôturer cette partie notez qu'il existe le mot clé **typeof** qui retourne une instance de la classe **System.Type**. Exemple d'utilisation

```
Type type = typeof(ClassExample);
```

Dans certains cas on est obligé de repartir le code d'un type sur plusieurs fichiers. Un cas d'utilisation pratique est lors de génération automatique du code du desgin par l'IDE Visual Studio. Pour déclarer un type partiel il faut utiliser le mot clé **partial** et ceci dans les différents fichiers. Les déclarations doivent être cohérentes (même nom, héritage, visibilité...). Voici un exemple, dans le premier fichier on déclare

```
public partial class ClasseFille : ClassExample
{
}
```

et dans le second

```
public partial class ClasseFille
{
    0 références
    public ClasseFille()
    {}
    0 références
    public void M()
    {}
}
```

La classe résultante contient l'héritage de classe **ClassExample**, le constructeur et la méthode **M**.

Complément commentaires

En C#, il existe une manière de grouper le code via les régions

```
#region usings
using l = LibExample;
#endregion

var instance = new l.ClassExample();
```

Le résultat est le suivant

```
+ usings
var instance = new l.ClassExample();
```

Notez qu'il est possible de faire une documentation technique directement sur le code via des balises XML par exemple


```

/// <summary>
/// Retourne le carré d'un nombre.
/// </summary>
/// <param name="d">Un nombre.</param>
/// <returns>Le nombre <paramref name="d"/> au carré.</returns>
0 références
public double Square(double d)
{
    return d * d;
}

```

Complément membres d'un type

Un champ peut être déclaré comme étant en lecture seule via le mot clé **readonly**, dans ce cas il n'est plus possible de le modifier en dehors des constructeurs. Voici un exemple

```

public class ClassExample
{
    readonly int field = 1;
    0 références
    ClassExample()
    {
        field = 2;
    }
}

```

Le mot clé **const** peut être aussi utilisé pour bloquer la modification même dans les constructeurs. Cependant tous les types ne peuvent pas être déclarés constants. Seules la valeur **null**, le booléen, les nombres et les chaînes de caractères peuvent l'être.

Il est possible d'avoir un comportement similaire pour les propriétés en utilisant le mot **init**. Exemple de déclaration

```

public class ClassExample
{
    0 références
    public int P { get; init; }
}

```

Ici on a remplacé le mot clé **set** de l'auto-implémentation par le mot clé **init** (on ne peut pas les utiliser en même temps). Si une propriété utilise ce type d'accesseur alors il est impossible de la modifier en dehors des constructeurs et de l'initialisation. Le code suivant est possible

```

var instance = new ClassExample { P = 10 };

```

On a vu précédemment les quatre genres d'un membre (champ, constructeur, méthode et propriété). Sachez qu'il existe un cinquième appelé le destructeur. C'est une méthode spéciale qui sera appelée automatiquement par le Garbage Collector lors de la libération de la mémoire. En général il est utilisé

pour libérer les ressources non gérées par le Garbage Collector comme certain type de mémoire, les pointeurs, les handles. Voici un exemple

```
public class ClassExample
{
    0 références
    ~ClassExample()
    {}
}
```

Ici on a déclaré le destructeur. Il a comme nom celui de la classe précédé par un tilde ~. Il n'accepte aucun paramètre. Seules les classes sont concernées par les destructeurs.

Il est possible de déclarer un champ de type évènement en ajoutant le mot clé **event**. La différence avec le champ est que l'affectation se fait via l'opérateur += et le champ doit être de type délégué. En général ce type de champ est utilisé pour souscrire des méthodes de rappel liées à un évènement comme le clic sur un bouton. Par exemple

```
public class ClassExample
{
    event DelExample field;
    1 référence
    public ClassExample()
    {
        field += M1;
        field += M2;
    }
    1 référence
    void M1()
    {}
    1 référence
    void M2()
    {}
}
```

Pour annuler une souscription on doit utiliser l'opérateur -=, par exemple

```
field -= M1;
```

Les types qui sont concernés par les champs sont les classes et les structures.

Comme avec les champs il est possible de déclarer une propriété comme étant un évènement en utilisant le mot clé **event** et en remplacement le mot clé **get** et **set** par **add** et **remove**. Voici un exemple

```

public class ClassExample
{
    event DelExample field;
    public event DelExample Field
    {
        add
        {
            field += value;
        }
        remove
        {
            field -= value;
        }
    }
}

```

Il existe une propriété spéciale qui se nomme **this** et qui accepte des arguments. Cette propriété s'appelle l'indexeur. Voici un exemple de déclaration

```

public class ClassExample
{
    int field = 1;
    0 références
    int this [int p]
    {
        get { return p * field; }
        set { field = p * value; }
    }
}

```

et un exemple d'utilisation

```

static void Main(string[] args)
{
    ClassExample inst = new ClassExample();
    int res = inst[10];
}

```

Les types qui sont concernés par les propriétés sont les classes, les structures et les interfaces.

L'utilisation des expressions lambda ne se limite pas aux fonctions anonymes mais peuvent être utilisées pour le corps des membres (constructeurs, méthodes et propriétés). Ce type de déclaration s'appelle expression body. Voici des exemples

```

public class ClassExample
{
    int field;
    0 références
    int P1 => 2 * field;
    0 références
    int P2
    {
        get => 3 * field;
        set => field = value;
    }
    0 références
    ClassExample(int f) => field = f;
    0 références
    int M(int v) => v * field;
}

```

L'utilisation des expressions bodies est parmi les bonnes pratiques.

Complément collections

Il est possible de déclarer un tableau multi-dimensionnels (genre matrice et autres). Voici un exemple d'un tableau à deux dimensions

```

int Method ()
{
    int[,] t = new int[,] { { -5, 5 }, { 1, -1 }, { 0, 0 }, { 6, -6 } };

    return t[0,0];
}

```

La propriété **Rank** donne la dimension d'un tableau. Dans l'exemple ci-dessus le rang du tableau est 2.

Pour accéder au dernier élément d'un tableau unidimensionnel on peut utiliser la propriété **Length** comme suit

```

var t = new[] { -5, -4, 0, 1, 2, 4 };
int dernierElement = t[t.Length - 1];

```

Il est possible de simplifier cette expression en utilisant le mot clé **^**, ainsi l'exemple précédemment devient

```

var t = new[] { -5, -4, 0, 1, 2, 4 };
int dernierElement = t[^1];
int avantDernierElement = t[^1];

```

Il est possible d'extraire un tableau à partir d'un autre en utilisant les rangées, on utilise dans ce cas le mot clé **..**. En fournissant l'indice de départ (inclus) et l'indice d'arrivée (exclus). Par exemple

```
var t = new[] { -5, -4, 0, 1, 2, 4 };
int[] subTab1 = t[1..4]; /* Tableau contenant les éléments -4, 0 et 1 */
int[] subTab2 = t[1..^2]; /* Tableau contenant les éléments -4, 0 et 1 */
```

Il existe d'autres types de collections comme **SortedList**, **Stack** ou encore **Queue**.

Il existe un autre type très utile qui est le dictionnaire. La classe associée est **Dictionary**. Cette classe permet de stocker des associations clé-valeur. La particularité de ce stockage est la rapidité d'accès en se basant sur les techniques de hachage. Voici un exemple

```
int Method ()
{
    Dictionary<string, int> d = new Dictionary<string, int>();

    d.Add("LUNDI", 2);
    d["JEUDI"] = 5;
    d["MARDI"] = 3;

    return d["JEUDI"];
}
```

La propriété **Count** retourne la taille du dictionnaire.

Complément héritage

Voici la liste des méthodes de la classe **object**

```
public class Object
{
    public Object();

    ~Object();

    public static bool Equals(Object? objA, Object? objB);
    public static bool ReferenceEquals(Object? objA, Object? objB);
    public virtual bool Equals(Object? obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string? ToString();
    protected Object MemberwiseClone();
}
```

La classe contient un constructeur et un destructeur. Deux méthodes statiques et cinq méthodes d'instances. La méthode **ToString** retourne une chaîne de caractère représentant le type. Le type à retourner est de type **string?**. C'est un type **string** qui doit être vérifié avant d'être utilisé (référence vers un type nullable). La méthode **Equals** permet de comparer un type avec un autre. Cette méthode est utile lors de l'utilisation de l'opérateur **==** ou lors du hachage (utilisation de la classe dictionnaire). La méthode **GetHashCode** permet de retourner un code entier représentant les données de l'objet. Ce code est utilisé lors du hachage.

La classe **System.Console** est une autre classe utilisée pour les projets « Console » et dont les deux méthodes statiques **Console.Write** et **Console.WriteLine** sont les plus utilisées.

Complément polymorphisme

Il est possible de déclarer une méthode (ou propriété) qui porte le même nom et signature qu'une méthode de la classe mère. Dans ce cas il est conseillé d'utiliser le mot clé **new** (mécanisme appelé masquage). Un exemple

```
1 référence
class ClassExample
{
    0 références
    public void M() {}
}

0 références
class ClasseFille : ClassExample
{
    0 références
    public new void M()
    {}
}
```

La méthode exécutée dépend du type de la déclaration utilisée. Voici un exemple

```
static void Main(string[] args)
{
    ClasseFille c = new ClasseFille();

    c.M(); // Execute ClasseFille.M
    ((ClassExample)c).M(); // Execute ClassExample.M
}
```

Ici les deux méthodes sont considérées comme deux méthodes séparées. Le code exécuté est donc fixe.

Complément passage d'un paramètre

Il est possible dans le cas où la valeur d'un paramètre est passée par valeur, de changer ce comportement de telle façon que le passage se fait par référence. Dans ce cas on doit utiliser les mots clés **ref** et **out**. La différence entre les deux est que le **out** nécessite l'affectation de la valeur avant de quitter la méthode. Exemple

```

static void Main(string[] args)
{
    int e = 2;

    RefModify(ref e);
    Console.WriteLine(e); // Affiche -2

    OutModify(out e);
    Console.WriteLine(e); // Affiche -3
}
1 référence
static void RefModify(ref int e)
{
    e = -2;
}
1 référence
static void OutModify(out int e)
{
    e = -3;
}

```

Parmi les bonnes pratiques de programmation l'utilisation du mot clé **ref** est déconseillée (perte de contrôle du flux). Il est possible de déclarer la variable à affecter par le **out** dans le même code de l'appel via le mot clé **var**. Exemple

```

static void Main(string[] args)
{
    OutModify(out var e);
    Console.WriteLine(e);
}
1 référence
static void OutModify(out int e)
{
    e = -3;
}

```

Lors de l'appel d'une méthode (ou constructeur) on doit spécifier les valeurs de tous les paramètres et dans le même ordre de déclaration. Cependant il est possible de ne pas spécifier toutes les valeurs si la méthode est déclarée avec des paramètres optionnels. Pour déclarer un paramètre optionnel il suffit de spécifier une valeur par défaut qui sera utilisée si le paramètre n'est pas spécifié. Par exemple

```

static int Multiply(int n, int by = 1, bool neg = false)
{
    return n * by * (neg ? -1 : 1);
}
0 références
static void Main(string[] args)
{
    Multiply(5, 2); // Résultat 10
}

```

Il est possible de spécifier la valeur d'un paramètre en utilisant directement son nom. C'est pratique dans le cas où il y a plusieurs paramètres optionnels. Par exemple

```
static int Multiply(int n, int by = 1, bool neg = false)
{
    return n * by * (neg ? -1 : 1);
}
0 références
static void Main(string[] args)
{
    Multiply(5, neg:true); // Résultat -5
}
```

Parfois on aimerait utiliser un nombre variable de paramètres, dans ce cas on doit déclarer le dernier paramètre de la méthode avec le mot clé **params**. Par exemple

```
static int Sum(int multiply, params int [] values)
{
    int sum = 0;

    foreach (var v in values)
    {sum += v;}

    return multiply * sum;
}
0 références
static void Main(string[] args)
{
    Sum(2, -1, 2, 3); // Résultat 8
}
```

Complément espace de noms

Dans le cas d'un conflit de noms, le nom complet permet de le résoudre. Cependant si l'espace de noms est long cette solution devient lourde. Dans ce cas l'utilisation des alias permet d'atténuer cette lourdeur

```
using l = LibExample;

var instance = new l.ClassExample();
```

Parfois il est impossible d'atteindre un espace de noms parce qu'il est masqué, alors le mot clé **global ::** permet de construire l'arborescence du **using** en repartant de la racine des espaces de noms. Le fichier généré par défaut (par Visual Studio) des espaces de noms globaux est construit en utilisant cette possibilité

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
```


Complément chaînes de caractères

Pour écrire une chaîne de caractères avec des caractères spéciaux tels qu'une tabulation ou un retour à la ligne on est obligé d'utiliser le caractère d'échappement « \ ». Un exemple

```
static void Main(string[] args)
{
    var str = "Une tabulation \t, un retour chariot \r et une fin de ligne \n";
    var path = "C:\\Windows\\System32";
}
```

Il est possible avec le symbole @ de désactiver l'interprétation du caractère d'échappement et ainsi écrire une chaîne de caractères plus facilement surtout dans le cas d'un chemin de fichiers sous Windows. Exemple

```
static void Main(string[] args)
{
    var path = @"C:\Windows\System32";
}
```

Il est possible d'utiliser l'opérateur + pour concaténer plusieurs chaînes de caractères. Pour plus de lisibilité il est possible d'utiliser l'interpolation des chaînes de caractères via le symbole \$. Exemple

```
static void Main(string[] args)
{
    var str = "La taille des arguments est " + args.Length + ".";
    var strInter = $"La taille des arguments est {args.Length}.";
}
```

Surcharge d'opérateurs

Le fait d'utiliser l'opérateur + avec les chaînes de caractères vient d'une particularité du langage C# qui est la surcharge des opérateurs. Sans entrer dans les détails de comment surcharger un opérateur, sachez que plusieurs opérateurs ont été surchargés dans le .NET par exemples les opérateurs +, -, ==, >, <, +=, -=, ++, --, les casts implicites et explicites...

Parmi les surcharges d'opérateurs pratiques on a + et == pour la classe **string**, ==, !=, +, -, >, >=, <, <= pour les structures **DateTime** et **TimeSpan**, les conversions entre les nombres (par exemple entre **long** et **int** ou entre **int** et **float**).

Point d'entrée d'un assembly

Certains types de projets ont besoin d'un point d'entrée. En général on écrit une méthode statique qui se nomme **Main** acceptant un tableau de chaînes de caractères représentant les arguments passés lors de l'exécution de l'assembly associé au projet. Voici le code du point d'entrée dans le cas d'un projet **Console**

```
internal class Program
{
    0 références
    static void Main(string[] args)
    {
        Console.WriteLine("Nombre d'arguments " + args.Length);
        Console.WriteLine("Premier argument " + args[0]);
    }
}
```

Un cas d'exécution peut être comme suit

```
C:\net6.0>ConsoleExample "Test arguments" 50
Nombre d'arguments 2
Premier argument Test arguments
```

Il est possible de ne pas déclarer ni la méthode **Main** ni sa classe en utilisant une possibilité du langage C# appelée « Instructions de niveau supérieur » ou « Top level statements » en anglais. Dans ce cas le même code se réduit à

```
Console.WriteLine("Nombre d'arguments " + args.Length);
Console.WriteLine("Premier argument " + args[0]);
```

Ce bout de code est écrit directement dans un fichier C# sans méthode ni classe. Bien sûr, on a droit qu'à un seul fichier par projet contenant ce type d'instructions.

Bonnes pratiques de nommage

Parmi les bonnes pratiques de nommage des types quel que soit leurs genres ils doivent commencer par une majuscule et par **I** et une lettre en majuscule dans le cas d'une interface. La suite du nom doit suivre la convention « Camel Case ». Par exemple **VoiciMaClasse** ou encore **IVoiciMonInterface**.

Aussi parmi les bonnes pratiques, chaque type est déclaré dans un fichier séparé qui portera le même nom.

Pour les champs, ils ne doivent pas commencer par une majuscule. Les méthodes, les propriétés et les valeurs des énumérations doivent commencer par une majuscule en suivant la convention « Camel Case ».

Les bonnes pratiques de nommage d'une variable locale sont les mêmes pour un champ (ne commencent pas par une majuscule).

Structures de contrôle

Les constructeurs, les destructeurs et les méthodes peuvent contenir du code. Ce code est appelé corps.

Au niveau d'un corps, il est possible de déclarer une variable locale en spécifiant le type et le nom, exemple

```
float Methode()  
{  
    int maVariable = 1;  
    const float pi = 3.14f;  
  
    return maVariable * pi;  
}
```

Il est possible de déclarer une variable en lecture seule via le mot clé **const**.

Une variable locale est visible uniquement dans son bloc de code. Un bloc de code est le code entouré par les accolades ouvrante et fermante {}.

Une méthode peut retourner une valeur avec le mot clé **return**.

On peut utiliser les mots clés **if**, **else if** ou **else** si on veut exécuter un bloc de code d'une façon conditionnelle, par exemple

```
float Methode(int c)  
{  
    const float pi = 3.14f;  
  
    if (c == 0)  
    {  
        return pi;  
    }  
    else if(c == 1)  
    {  
        return 2 * pi;  
    }  
    else  
        return 3 * pi;  
}
```

Dans le cas où le bloc de code peut se tenir sur une ligne alors il est possible d'utiliser l'opérateur ternaire ? dont voici un exemple

```
float Methode(int c)
{
    const float pi = 3.14f;

    float value = c == 0 ? pi : 2 * pi;

    return value;
}
```

Il y a une autre utilisation du même opérateur et ceci dans le cas où on veut invoquer une méthode si l'instance est non nulle (si c'est nulle aucune action à faire). Voici un exemple

```
static void Main(string[] args)
{
    ICloneable c = GetSource1();
    c?.Clone();

    c = GetSource2();
    c?.Clone();
}
1 référence
static ICloneable GetSource1()
{
    return null;
}
1 référence
static ICloneable GetSource2()
{
    return "A";
}
```

L'opérateur ?? permet lui de retourner la même valeur si ce n'est pas nulle et une autre valeur (à spécifier) si c'est nulle. Exemple

```
static void Main(string[] args)
{
    ICloneable c = GetSource1() ?? "B";
    c.Clone();

    c = GetSource2() ?? "C";
    c.Clone();
}
1 référence
static ICloneable GetSource1()
{
    return null;
}
1 référence
static ICloneable GetSource2()
{
    return "A";
}
```

Une autre façon d'écrire l'exemple du if mentionné précédemment en utilisant le mot clé **switch**

```

float Methode(int c)
{
    const float pi = 3.14f;
    float value = 0;

    switch (c)
    {
        case 0:
            return pi;
        case 1:
            {
                value = 2 * pi;
                break;
            }
        default:
            return 3 * pi;
    }

    return value;
}

```

Comme pour d'autres langages il est possible de faire des boucles en C# via les mots clés **for**, **while** et **do**. Voici des exemples

```

void Methode()
{
    int[] liste = new int[] { -2, 1, 5, 6 };
    int tripleSum = 0;

    for (int i = 0; i < liste.Length; i++)
    {tripleSum += liste[i];}

    int counter = 0;
    while(counter < liste.Length)
    {
        tripleSum += liste[counter];
        counter++;
    };

    counter = 0;
    do
    {
        tripleSum += liste[counter];
        counter++;
    } while(counter < liste.Length) ;
}

```

Références

https://fr.wikipedia.org/wiki/Microsoft_Visual_Studio

<https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/>

https://fr.wikibooks.org/wiki/Programmation_C_sharp

<https://perso.esiee.fr/~perretb/I3FM/POO1/dotnet/index.html>