haute école
neuchâtel berne jura
arc ingénierie
www.he-arc.ch

Semestre automne 2021

Revision .NET - CP1

Authors :
Simon Meier

Table of content :

1. Introduction

1.1 Pour faire genre

- .NET : cadriciel
- .NET/C# apporte une approche unifiée comme nouvelle base de développement.
- .NET runtime doit être installé sur la machine utilisateur.
- Paradigme orienté objet complet.
- Excellent design de la bibliothèque de base BCL.
- Indépendance des langages par la compilation en langage intermédiaire.
- Support pages Web Dynamique avec ASP.NET, Ajax, jQuery
- Support XML intégré.
- ORM -> Entity Framework -> BDD
- Assemblies : partage du code, gestion des versions, zero impact installation.

1.2 Basique sur le langage

- Syntaxe C++/Java
- Fortement typé
- Documentation par XML
- Garbage collection
- Possibilité d'utiliser des pointeurs et d'adresser directement la mémoire: mot-clé *[unsafe]*

```
// Exemple 1:
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}

// Exemple 2:
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}

// Exemple 3:
```

```
// You can also use an unsafe block
// to enable the use of an unsafe code inside this block.
unsafe
{
    // Unsafe context: can use pointers here.
}
```

- Support de propriétés et d'évènements:
  Déclaration de propriétés:

```
class MyClass
{
    private int x;
    public int X
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }

    // static
    private static int x;
    public static int Y
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }
}

// utilisation :
MyClass mc = new MyClass();
mc.X = 220; // set
int val = mc.X // get
```

:::info
Les propriétés d'une classe peuvent être héritées dans la classe dérivée.
:::
Déclaration d'évènement:
R>

```
using System;

namespace SampleApp
{
   public delegate string MyDel(string str);
```

```
   class EventProgram
   {
      event MyDel MyEvent;

      public EventProgram()
      {
         this.MyEvent += new MyDel(this.WelcomeUser);
      }
      public string WelcomeUser(string username)
      {
         return "Welcome " + username;
      }
      static void Main(string[] args)
      {
         EventProgram obj1 = new EventProgram();
         string result = obj1.MyEvent("Tutorials Point");
         Console.WriteLine(result);
      }
   }
}
```

- Multi-plateformes
- Le Gaming (Unity, OpenGL, GoDot, Metal)
- Code très performant

2. Assemblage (*Assembly*)

Un assembly est la sortie compilée du code, généralement une DLL, mais un .exe est également un assembly.

Il s'agit de la plus petite unité de déploiement pour tout projet .NET.

L'assembly contient généralement du code .NET en MSIL (langage Microsoft Intermediate) qui sera compilé en code natif ("JITted" - compilé par le compilateur Just-In-Time) la première fois qu'il est exécuté sur une machine donnée.

Ce code compilé sera également stocké dans l'assembly et réutilisé lors des appels suivants.

L'assembly peut également contenir des ressources telles que des icônes, des bitmaps, des tables de chaînes, etc.

En outre, l'assembly contient également des métadonnées dans le **manifeste de l'assembly** - des informations telles que le numéro de version, le nom fort, la culture, les assemblys référencés, etc.

Dans 99% des cas, un assemblage équivaut à un fichier physique sur le disque.

Dans un assembly multifichier, il n'y aurait toujours qu'un seul manifeste Assembly dans un `.dll` ou `.exe` et le code MSIL dans plusieurs fichiers de netmodule).

3. C#, basiquement

- un seul fichier code source (.cs), pas de header.
- structuré en blocs { }
- imbrication des blocs (nested)
- bloc peut contenir zéro une, ou plusieurs instructions statements.
- point d'entrée:

```
public static void Main()
{
    // code
}
```

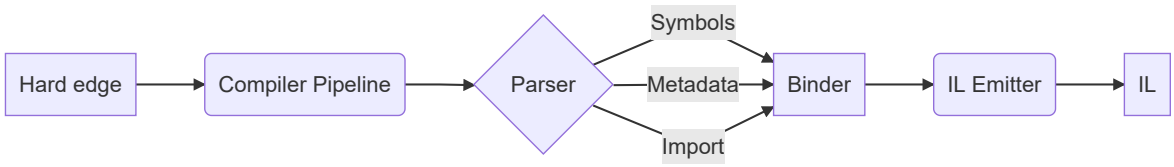
- langage très fortement typé.

3.1 Namespaces

```
System.IO.FileStream
dataFile ;;// fully qualified named
using
System.IO ;;// directive using d'un namespace
FileStream
dataFile ;
using static
System.Console ;;// v
WriteLine
("Hello World");// au lieu de Console. WriteLine
```

- Création automatique d'un espace de nom homonyme au projet.
- Création d'espaces de noms.
- Accès aux éléments dans des espaces de noms:
  - accès explicite (fully qualified name)
  - directive `using`
- Accès aux méthodes d'une classe statique abrégé avec `using static`.

3.2 Compilateur C#, la Roslyn



:::info
Pour compiler un code C# en `Intermediate Language`, on invoque le compilateur **csc.exe**.
:::

Syntaxe:

- `csc [options] HelloWorld.cs /t :target /reference:ref1.dll`
  Exemple:
- `csc /t:exe /reference:assemb1.dll HelloWorld.cs`

3.3 Type de variables

Value	Reference
La variable contient ses données	Variable dans la pile qui référence un objet du tas géré
L'affectation copie les données	L'affectation copie la référence, pas l'objet
Durée de vie -> } (leur portée)	Durée de vie non déterministe (garbage collection)
Existence dans la pile stack	La déclaration de la variable ne crée pas l'objet null
Opérateurs de comparaison contenu(values)	Tous les champs de l'objet sont à 0 Null
Ne peut pas prendre la valeur null	Opérateurs de comparaison de références

Value: int float bool enum struct
Reference: string, classes, tableaux

3.3.1 `var`, `decimal` et `enum`

- 1. `var` permet le typage implicite `statique` de variables **locales**.
- 2. `decimal` a~30 digits. Lent.
- 3. `enum` définis dans un namespace (hors classe); usage recommandé.

Exemple d'enum:

```
enum Months
{
  January,    // 0
  February,   // 1
  March=6,    // 6
  April,      // 7
  May,        // 8
  June,       // 9
  July        // 10
}
```

3.3.2 `struct`

- `struct` est un type **value**: allocation automatique dans la pile (+performant).
- pas d'héritage (le C tu connais)
- pas de définition possible du constructeur par défaut.
- définition dans un namespace où une classe.
- on constructeur **doit** initialiser tous les champs.
- pas de définition de desctructeur possible.
- `struct` hérite uniquement de `System.Object`.
- pas d'initialisation de champ à sa déclaration.
  Exemple:

```
public struct Data
{
    public List<object> data;
    public String category;

    public Data(String data, String category)
    {
        this.category = category;
        this.data = new List<object>();
    }
}
```

3.4 Opérateurs et expressions

Opérateurs	Opération
coalescent `??=`:	null-coalescing operator ?? retourne la valeur de l'opérande gauche si elle est pas nulle; sinon elle évalue celle de droite et retourne son résultat.
assignement coalescent `??=`:	null-coalescing assignment operator `??=` *assigne* la valeur à l'opérande de **droite** à celle de **gauche** seulement si l'opérande est évaluée à `null`
Exemple 1:	

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
```

Exemple 2:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

:::info
The null-coalescing operators sont associatif en commençant par la droite;

```
a ?? b ?? c
d ??= e ??= f
==
a ?? (b ?? c)
d ??= (e ??= f)
```

C'est utile si on travaille avec des `null-conditional operators ?. and ?[]` où des `nullable value types` comme ça on peut leur donner une valeur spécifique dans le cas où le type est `null` .
Exemple simple:

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b);  // output: -1
```

:::

Les autres opérateurs usuels, la flemme: liste

3.5 Instructions de contrôle

- `if` : pas de conversion implicite. (bool <- int)
- `foreach` : on connaît.
- `for, while (){}, do { } while (), goto , break, continue, return` : bref
- `switch` : sympa
  Exemple:

```
object obj = new Developer {
    FirstName = "Neil",
    YearOfBirth = 1980
    };

string favoriteTask;

switch (obj) // Since C# 7.0, any type is supported here
{
  case Developer _: // Type pattern with discard (_)
    favoriteTask = "Write code";
    break;
  case Manager _:
    favoriteTask = "Create meetings";
    break;
  case null: // The null pattern
    favoriteTask = "Look into the void";
    break;
  default:
    favoriteTask = "Listen to music";
    break;
}
```

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException(
           "Invalid string value for command",
           nameof(command)
       ),
   };
```

3.6 Méthodes

- Passage des paramètres par valeur (copie)
- Surcharge possible.
- Valeurs par défaut.
- **Modificateurs ret et out** pour les paramètres.
- Méthodes génériques (`Swap<T>`)
- `ref` : donne la possibilité de passage d'un paramètre *par référence* (puisque le passage se fait par valeur par défaut);
    - préfixer le type du paramètre par `ref` dans la déclaration **et** à l'appel de la méthode.
    - utilisable sur tous les types (value où reference)
      Exemple simple

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

Exemple: surcharge de méthode

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

- `out` : passe l'argument par référence et force la méthode à lui assigner une valeur.
    - préfixer le type du paramètre par `out` dans la déclaration **et** à l'appel de la méthode.
    - la variable ne doit pas être `const`
    - **tous** les chemins d'exécutions doivent affecter une valeur;
    - pas d'obligation d'initialiser `param` avant l'appel.
      Exemple:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

- déclaration de variables out *inline* possible:

```
if (int.TryParse(input, out int result)) WriteLine(result);
else WriteLine("alkdjfléaskjdf");
```

- `params Type []` : permet au dernier paramètre d'avoir une **cardinalité variable**;
    - du coup on peut spécifier un nombre variable d'argument pour le paramètre.
    - le paramètre doit être un tableau 1D.
      Exemple

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
```

```
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

3.7 Déléguées

Type qui permet de référencer une méthode d'une classe; similaire au pointer de fonction en C++ et C, sans tous les désavantages (poo, sûre, bref la vie). Mécanisme pour utiliser une méthode comme un objet. Pas d'héritage possible depuis la classe `Delegate` et les `delegate` sont `sealed`.

- `delegate`, `Action<T>`, `Func<T>`: équivalent
- instancier un variable `delegate`
- affecter une méthode à la variable
- l'invoquée.
  Exemple:

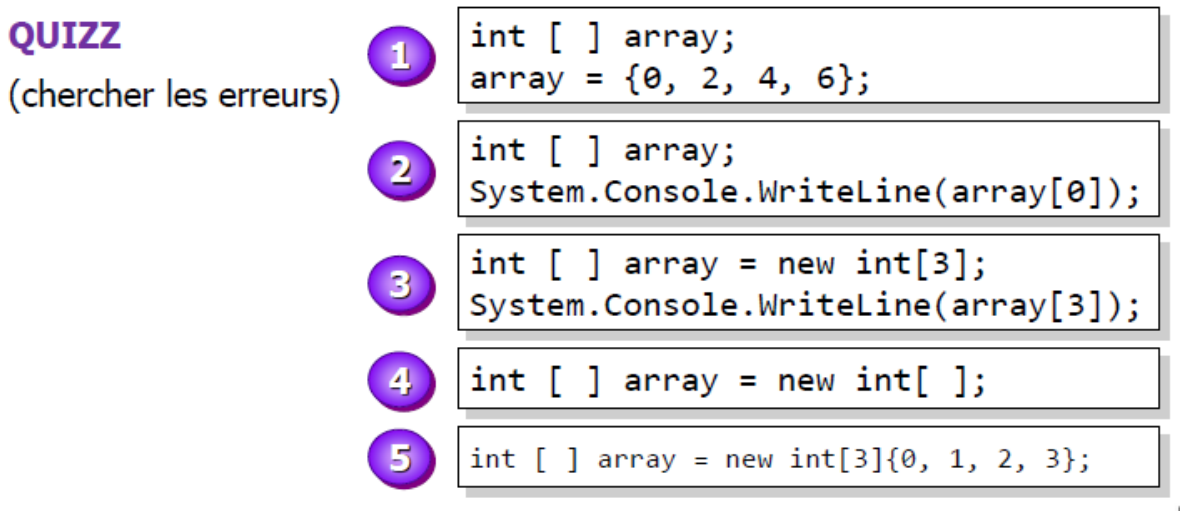
```
public delegate void Del(string message);

public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}

// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

3.8 Tableaux

Objet de la classe `System.Array`, la déclaration n'initialise pas le tableau, il faut le créer soit même avec des valeurs.

- pas de contrôle d'indice.
- type : **reference**
- méthodes: Clear, Clone, GetLength(int), IndexOf, Sort
- propriétés: Rank, Length.
- accès à tous les éléments avec `for, foreach`.
- statique.
- un peu plus performant que les listes.
- tableaux de tableaux : `int[][][][]`

QUIZZ:




```
// 1. correction: pas d'assignation.
int[] array;
array = new int[4]{1, 2, 3, 4};

// 2. correction: pareil et taille zéro (mini 1)
int[] array1 = new int[1] {0};
System.Console.WriteLine(array[0]);

// 3. correction: index out of bound évident
int[] array2 = new int[4];
System.Console.WriteLine(array2[3]);

// 4. correction: si on initialise on a besoin d'une taille
int[] array = new int[10];

// 5. bref ...
```

3.9 Gestion des exceptions

Après un bloc `try`, quelle est la différence entre une instruction placée dans un bloc `catch{…}` et la même instruction dans un bloc `finally`:

:::success
Le bloc `catch` n'est exécuté que si une `exception` est levée dans le bloc `try`. Le bloc `finally` est exécuté toujours après le bloc `try` qu'une exception soit levée ou non.
:::

:::danger
Plusieurs blocs catch peuvent être utilisés pour attraper différentes classes d'exceptions. L'exécution normale (lorsqu'aucune exception n'est lancée dans le bloc try) continue après le dernier bloc catch défini dans la séquence. Les exceptions peuvent être lancées (throw) ou relancées dans un bloc catch
:::

3.10 Le type `string`

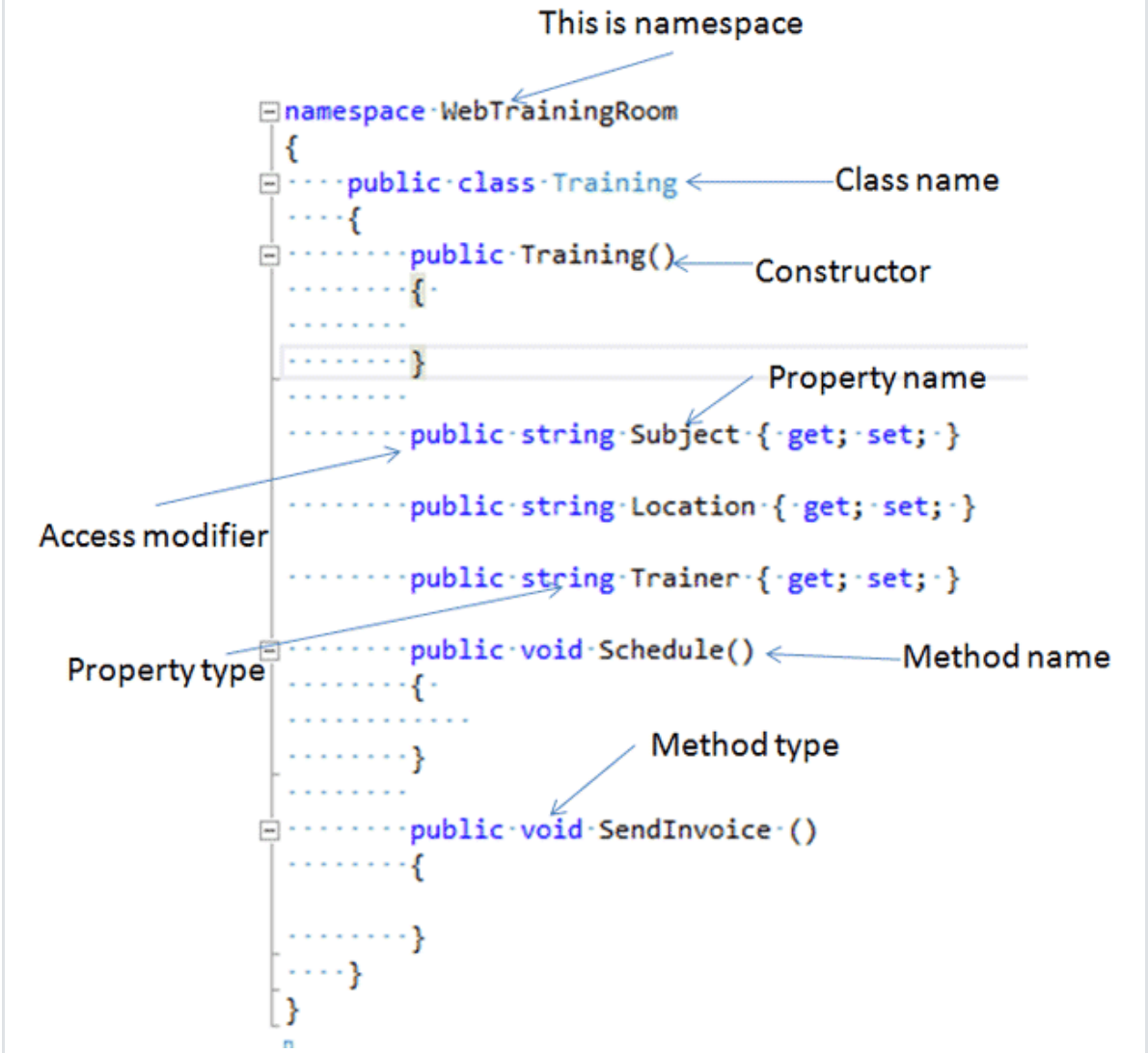
- constitué d'une chaîne de caractères Unicode (2bytes par caractère)
- type par référence
- Immuable

- accès indexé [] `read-only`
- propriété Length
- valeur littérales:
  - interpolées: $"{x},{y}"
  - verbatim: @"sur plusieurs lignes"
  - classique: "x:{0}",x
- Méthodes:
  - Insert(), Split(), Copy(), Concat(), Format(), Trim(), ToUpper(), ToLower()
- Equals(): comparaison par valeur
- Compare(): plus de comparaison, option avec ou sans casse, tri*.
- `==` et `!=` sont surchargés pour `String`.

4. Programmation Orientée Objet

4.1 Déclaration d'une classe

Schema



- Accesser:
  - `internal` : classe utilisable depuis la même assembly.
  - `public`: classe utilisable depuis le même ou d'un autre assembly

    - `abstract`: classe non instanciable (peut contenir des méthodes abstract)
    - `static`: classe non instanciable , que des membres `static` , est `sealed`.
    - `partial`: définissable dans plusieurs fichiers.
    - `sealed`: ne peut pas être héritée / spécialisée.
    - `unsafe`: peut contenir du code où la mémoire n'est pas forcément gérée.
- Il faut utiliser le mot-clé `new` pour créer un objet. La déclaration ne suffit pas.

4.2 Champs

variable membre d'une classe, initialisé par défaut (`0, null , false`)

Modificateurs:

- `static`, `new`, `unsafe`, `readonly`, `const`, `volatile`, `public`, `internal` , [`private`]*(par défaut), `protected`

Exemple:

```
class BankAccount
{
    static public double InterestRate = 0.0125; // bof
    readonly string name ;// private , init -> ctor
    private double amount ;// = 0.0 par défaut
    const double gold = 1.61803398875;// valeur requise

    public BankAccount (string name )
    {
        this.name = name;// readonly init ds ctor
    }
}
```

4.2 Propriétés

Une propriété ressemble à un champ mais, en interne, elle contient de la logique.

- property accessors:
    - `get`: exécuté quand la valeur de la propriété est lue doit renvoyer une valeur du type de la propriété.
    - `set`: exécuté quand la propriété est cible d'une affectation mot clef `value`

Exemple:

```
class Car
{
    int nbDoors ; // champ de support private
    public int N bDoors
    {
        get { return n bDoors; }
        set{ if (( value == 3 || value ==5)) n bDoors = value ; }
    }
}

Car myCar = new Car()
NbDoors = myCar.NbDoors = 9; // nbDoors reste à 5
```

5.2 Propriétés

- valeur par défaut
- auto-implémentation

```
public int Score { get ; set ; } = 10
```

- *read-only* (write-only)

```
public class Vector{
    public double X { get ;  }        // read-only
    public double Y { get ;  }        // read-only
    public Vector(float x, float y) {
           this.X = x ;
            Y = y ;
    }
}
```

- `value`
- *backing field*

```
public class Time{
  private int seconds;          // backing field(bf)
  public  int Seconds  {        // propriété
      get { return seconds;  }
      set { seconds = value; }
  }
  public int Minutes{  //propriété read-only sans (bf)
      get { return seconds/60;}
  }
}
```

- snippets *prop propg...*
- niveaux d'accessibilité

4.3 Constructeurs

- Constructeur par défaut synthétisé.
- Surcharge possible du constructeur.
- Appel d'un autre constructeur dans une liste d'initialisation:
  Exemple1:

```
class Date
{
    public Data(int y, int m, int d)
    { ... }
    public Data(int year)
    { ... }

    public Data() : this(1970, 1, 1) { ... }
}
```

Exemple2:

```
public Point2D(double x, double y)
{
    // Contracts

    Initialize(x, y);
}

public Point2D(Point2D point)
{
    if (point == null)
        throw new ArgumentNullException("point");

    // Contracts

    Initialize(point.X, point.Y);
}
```

```
private void Initialize(double x, double y)
{
    X = x;
    Y = y;
}
```

- Les membres `readonly` doivent être initialisés à l'intérieur du constructeur.

4.4 Constructeur statique

Utilité:

- peut être utilisé pour initialiser les champs `static`
- sera **toujours** appelé avant le constructeut d'instance.

Restrictions:

- ne peut pas être appelé explicitement
- pas de modificateur d'accès
- pas de paramètre

Exemple:

```
class Test
{
    public static int Year { get; }
    static Test() { Year = System.DateTime.Now.Year; }
}
```

4.5 Finaliseur

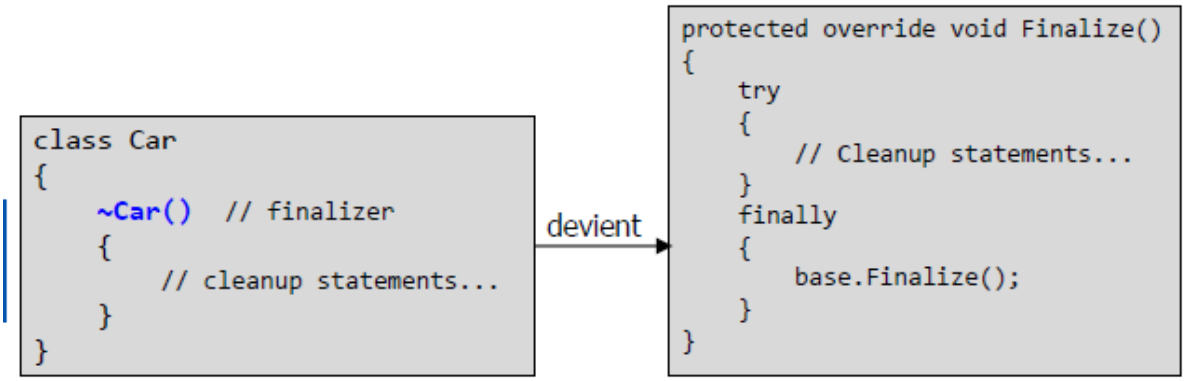
Le finaliseur est exécuté quand le garbage collector récupère les ressources (imprévisible).

Il remonte automatiquement la hiérarchie des classes `base.Finalize()`.

Syntaxe:

- pas de modificateur d'accès, de type de retour ni de paramètre

Comparaison C++ et C#:



4.5 Indexers

Les *Indexers* facilitent l'accès aux éléments d'une liste, collection, tableau encapsulé dans une `class` ou `struct`.

Une classe peut déclarer un ou plusieurs indexeurs avec la propriété `this`. Ce mécanisme est appelé *surcharge d'indexeurs*.

Exemple:

```
class Phrase
{
    string[] mots="Un tien vaut mieux que deux ".Split();
    public string this [int indiceMot]; // indexeur
    {
        get { return indiceMot ]; }
        set { indiceMot ] = value; } // peut être omis
    }
}

// use case:
Phrase dicton=new Phrase();
Console.WriteLine(dicton[1]);// tien
dicton[5] = "rien";
Console.WriteLine(dicton[ 5]);// rien
```

4.6 Surcharge d'opérateurs

- Les opérateurs surchargés en C# sont déclarés `public` et `static`.

Syntaxe: `public static returnValue operator op ( object1 [, object2])`

Exemple:

```
public static Time operator+(Time t1, Time t2)
{
    int newHours = t1.hours + t2.hours;
    int newMinutes = t1.minutes + t2.minutes;
    return new Time(newHours , newMinutes);
}
```

4.6.1 Opérateurs de comparaison

Doivent être **surchargés par paire**:

- `<` et `>`
  Exemple:

```
public static bool operator <(Box box1, Box box2)
{
    bool status = false;

    if (box1.volume < box1.volume)
    {
        status = true;
```

```
    }
    return status;
}
public static bool operator >(Box lhs, Box rhs)
{
    bool status = false;

    if (box1.volume < box1.volume)
    {
        status = true;
    }
    return status;
}
```

- `<=` et `>=`
  Exemple:

```
public static bool operator <=(Box box1, Box box2)
{
    bool status = false;

    if (box1.volume <= box2.volume)
    {
        status = true;
    }
    return status;
}
public static bool operator >=(Box box1, Box box2)
{
    bool status = false;

    if (box1.volume >= box2.volume)
    {
        status = true;
    }
    return status;
}
```

- `==` et `!=`
  Exemple:

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;
```

```
        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

Il est *nécessaire* de vérifier la profondeur de la comparaison. C'est pourquoi il faut aussi redéfinir les méthode `GetHashCode()` et `Equals` en surchargeant les opérateurs `==` et `!=`.

4.6.2 Opérateurs composés (`+=`, `^=`, ...)

Ils ne peuvent pas être surchargés explicitement.
D'ailleurs, en surchargeant `+` par exemple, pas besoin de surcharger `+=` car la surcharge est *implicite* (oui je me répète).
Autant simplement dire que les opérateurs composés ne peuvent pas être surchargés: source

4.6.3 Opérateur d'indexation `[]`

Il n'est pas considéré comme un opérateur surchargeable, et donc ne peut pas être surchargé directement, mais par l'utilisation d'`indexeurs` (cf. indexeurs) indexers.

4.6.4 Opérateur de conversion

Qualifié avec les mots-clés `implicit` ou `explicit`, permet de faire de la conversion d'objet sans passer par des méthodes redondantes pour passer les données d'un objet à un autre dans chaque définition de classe.

Exemple pour l'opérateur `implicit`:

```
public static implicit operator AuthorDto(Author author)
{
    AuthorDto authorDto = new AuthorDto();
    authorDto.Id = author.Id.ToString();
    authorDto.FirstName = author.FirstName;
    authorDto.LastName = author.LastName;
    return authorDto;
}

static void Main(string[] args)
{
    Author author = new Author();
    author.Id = 10;
    author.FirstName = "Machin";
```

```
    author.LastName = "Truc";
    // utilisation de l'operateur implicit
    AuthorDto authorDto = author;
    Console.ReadKey();
}
```

Exemple pour l'opérateur `explicit`:

```
public static explicit operator AuthorDto(Author author)
{
    // même code que plus haut, seul le mot-clé change
}

static void Main(string[] args)
{
    Author author = new Author();
    ...
    // utilisation de l'operateur explicit
    AuthorDto authorDto = (AuthorDto)author;
    Console.ReadKey();
}
```

4.6.5 Opérateur `is`

Indique si une variable est compatible avec le type spécifié:

- is : return bool

4.6.6 Opérateur `as`

Essaie de réaliser une conversion d'un objet:

- as
  En cas d'erreur: `return null`, donc pas d'exception.

5. Héritage en C#

C# et .NET prennent uniquement en charge l'héritage **simple**. C'est-à-dire qu'une classe ne peut hériter que d'**une** seule classe.

Toutefois, l'héritage est transitif, ce qui permet de définir une hiérarchie d'héritage pour un ensemble de types. En d'autres termes, le type `D` peut hériter du type `C`, qui hérite du type `B`, qui hérite du type de classe de base `A`. Étant donné que l'héritage est transitif, les membres de type `A` sont disponibles pour le type `D`.

Caractéristiques générales:

- Chaque classe peut hériter d'**une** classe et de plusieurs *interfaces*.
- Toute classe hérite de la classe *racine* **object**.
- Il est possible d'affecter une référence de n'importe quel type à une référence de type `object` -> `boxing`, `is`, `as`.
- Une classe dérivée **ne peut pas être plus accessible** que sa classe mère.

Tous les membres d'une classe de base ne sont pas hérités par les classes dérivées. Les membres suivants ne sont pas hérités :

- Les **constructeurs statiques**, qui initialisent les données statiques d'une classe.

- Les **Constructeurs d'instance**, que vous appelez pour créer une nouvelle instance de la classe. Chaque classe doit définir ses propres constructeurs.
- Les **finaliseurs**, qui sont appelés par le récupérateur de mémoire du *runtime* pour détruire les instances d'une classe.

Bien que tous les autres membres de classe de base sont hérités par les classes dérivées, leur visibilité dépend de leur accessibilité.

L'accessibilité d'un membre affecte sa visibilité pour les classes dérivées de la manière suivante :

- Les membres `private` sont visibles uniquement dans les classes dérivées qui sont imbriquées dans leur classe de base. Sinon, ils ne sont pas visibles dans les classes dérivées.
- Les membres `protégés` sont visibles uniquement dans les classes dérivées.
- Les membres `internes` sont visibles uniquement dans les classes dérivées qui sont trouvent dans le même assembly que la classe de base. Ils ne sont pas visibles dans les classes dérivées situées dans un autre assembly à partir de la classe de base.
- Les membres `public` sont visibles dans les classes dérivées et font partie de l'interface publique de la classe dérivée. Les membres `public` hérités peuvent être appelés comme s'ils étaient définis dans la classe dérivée.

Les classes dérivées peuvent également substituer les membres hérités en fournissant une implémentation alternative.

Pour être en mesure de *substituer* un membre, le membre de la classe de base doit être marqué avec le mot-clé `virtual`. Par défaut, les membres de classe de base ne sont pas marqués comme `virtual` et ne peut pas être substitués.

Une tentative de substituer un membre non virtuel, comme dans l'exemple suivant, génère l'erreur de compilateur CS0506 : « : impossible de substituer le membre hérité , car il n'est pas marqué comme `virtual`, `abstract` ou `override`.

- Les méthodes virtuelles sont polymorphiques.
- Les méthodes virtuelles ne peuvent ni être `static` ni `private`.
- Une méthode `virtual` peut avoir une implémentation, mais pas une méthode `abstract`.
- Pour `override`, la méthode de la classe mère doit être virtuelle.

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

5.1 Mot clé `base`

Le mot-clé `base` est utilisé pour accéder aux membres de la classe mère depuis l'intérieur de la classe fille.

- appeler une méthode de la classe de base qui a été redéfinis par une autre méthode.
- spécifier quel constructeur de la classe mère doit être appelé lors de la création d'instance de la classe dérivée.

Une *base class access* est permis seulement dans le constructeur, une méthode d'instance où accesseur de propriété.

C'est une erreur d'utiliser le mot-clé `base` dans une méthode statique.

Exemple 1:
utilisation de `base` pour appeler une méthode d'instance et l'utiliser dans la classe fille.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

Exemple 2:
Spécification du constructeur de la classe mère qui doit être appelé lors de la création d'instances de la classe fille.

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base() {    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i) {    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

5.2 Mot-clé `new`

En cas d'utilisation comme un modificateur de déclaration, le mot clé `new` masque explicitement un membre qui est hérité d'une classe de base. Lorsqu'on masque un membre hérité, la version dérivée du membre **remplace** la version de classe de base. Cela suppose que la version de la classe de base du membre est visible, car elle serait déjà masquée si elle était marquée comme `private` ou, dans certains cas, `internal`.

Il permet ainsi de bloqué le polymorphisme et de résoudre des collisions de nom dans le code.

Il est également possible d'utiliser `new` pour créer une instance d'un type ou l'utiliser comme une contrainte de type générique.

Pour masquer un membre hérité, déclaration dans la classe dérivée en utilisant le même nom de membre, puis modification à l'aide du mot-clé new.
Exemple:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

Dans cet exemple, `BaseC.Invoke` est masqué par `DerivedC.Invoke`. Le champ `x` n'est pas affecté parce qu'il n'est pas masqué par un nom semblable.

5.3 Mot-clé `sealed`

Il n'est pas possible d'hériter d'une classe `sealed`.

- Une classe scellée peut être plus rapide à l'exécution.
- .NET contient beaucoup de classes scellées (String, StringBuilder, ...)

5.4 Interfaces

Une `interface` ne fait que décrire une liste de méthodes, sans implémentation. Le code de ces méthodes est fourni par les classes qui implémentent l'`interface`.

Exemple:

```
public interface IAffichable
{
    /*
        Liste des méthodes que doivent posséder toutes les classes
        implémentant l'interface IAffichable :
    */
    void Afficher();
    void Afficher(string message);
}
```

- elle peut posséder: méthodes, propriétés, events, indexeurs.
- son nom commence par un `I`.
- ses méthodes n'ont pas de modificateur.
- elle peut hériter d'autres interfaces.
- une classe **doit** implémenter toutes les opérations des interfaces héritées.

Implémentation implicite:
L'appel aux méthodes de l'interface peut se faire en utilisant une référence à un objet Personne ou à une interface IAffichable :

Exemple:

```
Personne p = new Personne(nom, prenom);
p.Afficher();  // Appel via un objet Personne

IAffichable a = (p as IAffichable);
a.Afficher();  // Appel via une référence à l'interface
```

Implémentation explicite:

L'implémentation explicite se fait en donnant explicitement le nom de l'interface correspondant à la méthode implémentée.

L'interface IClonable déclare une méthode retournant un objet identique à l'objet utilisé:

```
public interface ICloneable
{
    object Clone();
}
```

L'implémentation explicite pour la classe Personne est la suivante :

```
using System;

public class Personne : IClonable
{
    private string nom, prenom;

    public Personne(string nom, string prenom)
    {
        this.nom = nom;
        this.prenom = prenom;
    }

    object ICloneable.Clone()
    {
        return new Personne(nom, prenom);
    }
}
```

La méthode ne déclare pas sa visibilité car c'est la même que celle de l'interface.

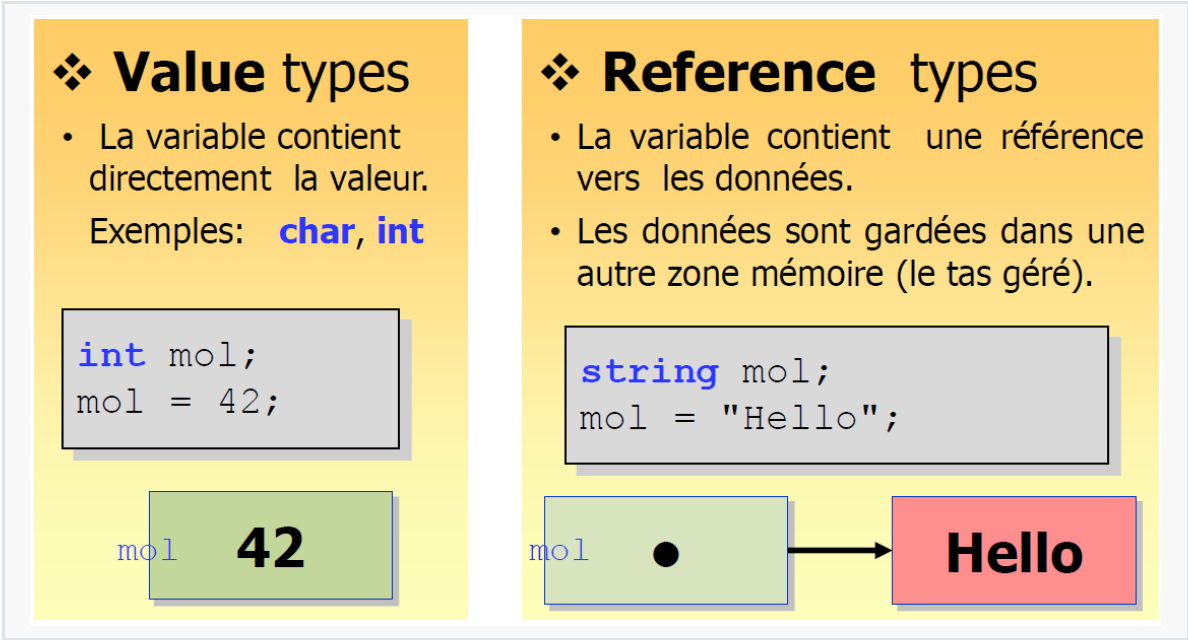
L'utilisation de la méthode ne peut alors se faire que par une référence à l'interface:

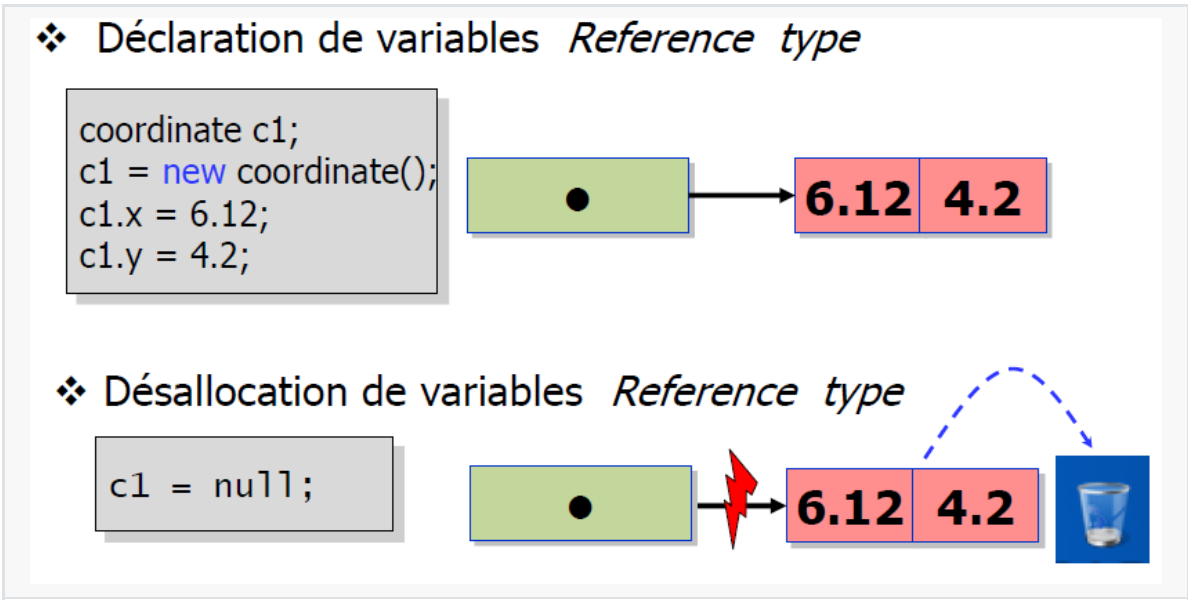
```
Personne p = new Personne(nom, prenom);
object p2 = (p as ICloneable).Clone();  // Appel via une référence à l'interface
```

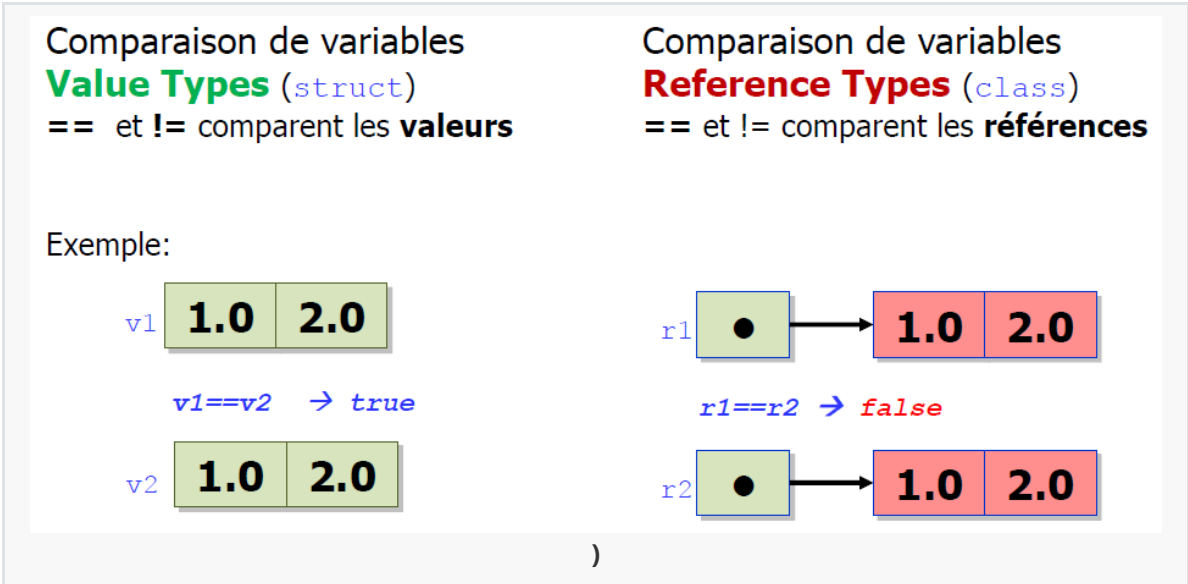
5.5 Mot-clé `abstract`

- Seule une classe `abstract` peut déclarer une méthode `abstract`.
- Une méthode `abstract` ne peut pas avoir d'implémentation.
- Une méthode `abstract` est virtuelle par définition.
- Une classe abstraite n'est pas instanciable.

6. Typage valeur et référence







)

7. Conversions

- Conversions implicites : `doule d = 1.23f;`
- Conversions explicites : `(int)12.34;`

- **upcast**:
  - implicite ou explicite.
  - réussit toujours: `object obj = ...`
- **downcast**:
  - cast explicite requis: `(Subclass)obj`.
  - vérifie le type réel de la référence -> throws InvalidCastException.
- Classe System.Convert permet de faire de la conversion entre types de bases.

8. Cycle de vie d'un objet

On ne peut pas détruire explicitement des objets.
Le **garbage collector** est chargé de la destruction des objets non référencés lorsque la mémoire disponible devient basse. Il trouve les objets inaccessibles et les dértuit:

- Exécution du code du destructeur (s'il existe).
- Finalisation (recyclage en mémoire brute disponible sur le tas avec .Finalize())

8.1 IDisposable

- Hériter de l'interface `IDisposable` et implémenter la méthode `Dispose()` afin qu'elle libère les ressources.
- `GC.SupressFinalize(this)` indique au garbage collector de ne plus appeler `Dispose()`.
- Faire en sorte que des appels multiples de `Dispose()` soient possibles.
- Ne pas utiliser une ressource réclamée.

```
public void Dispose()
{
    // Dispose of unmanaged resources.
    Dispose(true);
    // Suppress finalization.
    GC.SuppressFinalize(this);
}
```

8.1.1 Implémentation du pattern Dispose

Doc de microsoft:

- A Dispose implementation that calls the Dispose(bool) method.
- A Dispose(bool) method that performs the actual cleanup.
- Either a class derived from `SafeHandle` that wraps your unmanaged resource (recommended), or an override to the `Object.Finalize` method. The `SafeHandle` class provides a finalizer, so you do not have to write one yourself.

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

class BaseClassWithSafeHandle : IDisposable
{
    // To detect redundant calls
    private bool _disposedValue;

    // Instantiate a SafeHandle instance.
    private SafeHandle _safeHandle = new SafeFileHandle(IntPtr.Zero, true);
```

```
    // Public implementation of Dispose pattern callable by consumers.
    public void Dispose() => Dispose(true);

    // Protected implementation of Dispose pattern.
    protected virtual void Dispose(bool disposing)
    {
        if (!_disposedValue)
        {
            if (disposing)
            {
                _safeHandle.Dispose();
            }

            _disposedValue = true;
        }
    }
}
```

9. Généricité

Pour créer des méthodes et classes indépendantes d'un type:

- réutilisabilité
- performance
- sécurité du typage

Classes de collection:

- List, Queue, Stack, Dictionary<Tkey, Tvalue>
- -> interfaces, classes, méthodes, événements, délégués générique.
- Les classes génériques peuvent être contraintes (conditions sur le type).
    - Type de contraintes:

Contrainte	Description
where T: struct	*T doit être **un type par valeur** (VT)*
where T: class	*T doit être **un type par référence** (RT)*
where T: IToto	*Contrainte d'interface: T doit implémenter l'interface Itoto*
where T: CToto	*Contrainte de classe: T doit dériver de la classe de base Ctoto*
where T: new()	*Contrainte de constructeur. T doit avoir un constructeur par défaut*
where T1: T2	*« Naked type constraint »: T1 doit dériver d'un type générique T2*

Exemple 1 de contrainte:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>,
new()
{
    // ...
}
```

- une clause `where` distincte pour chaque type générique.
- les contraintes `struct` ou `class` s'excluent et doivent apparaître en premier.
- La contrainte de constructeur new() doit apparaître en dernier.

Exemple 2 de contrainte:

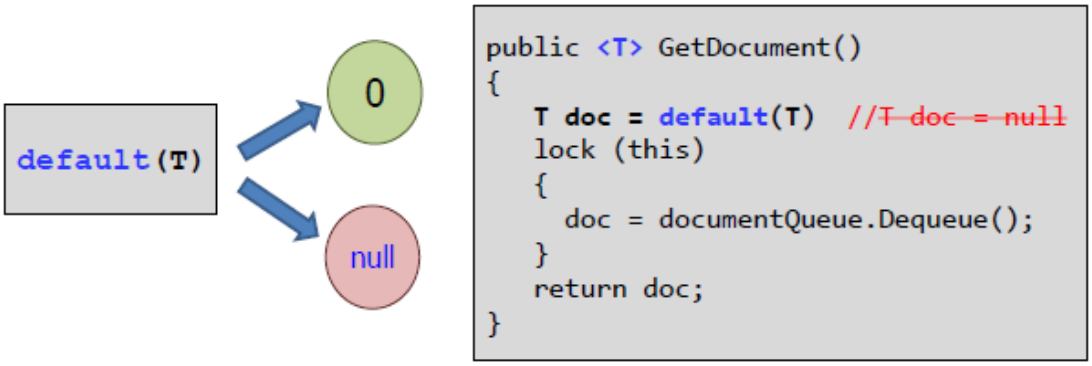
```
static void fonction <T1, T2> (T1 object, T2 valeur)
    where T1:class, Idisposable, new()
    where T2:struct
```

Exemple d'utilisation de `List<T>`:

```
List<string> dinosaurs = new List<string>();
dinosaurs.Add("Tyrannosaurus");
dinosaurs.Insert(2, "Compsognathus");
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}
```

Il est interdit d'affecter `null` à un type générique car celui ci peut instancier une `value` type pour lequel cette valeur est *illicite*.

Pour contourner ce problème, le mot clé `default` peut être utilisé:

- 

9.1 Interfaces génériques

Interfaces définissant des méthodes avec des paramètres génériques.

Interface	Description
IComparable <T>	`.Compare`
IEnumerable <T>	Requis pour `foreach`, définit `GetEnumerator()`
ICollection<T>	`Count`, `CopyTo()`, `Add()`, `Remove()`, `Clear()`
IList<T>	définit un indexeur; `Insert()`, `RemoveAt()`
ISet<T>	Unions, intersections, contrôle de chevauchement

9.2 Structures générique

Similaire aux classes génériques, exceptées qu'elles n'ont pas d'héritages.

9.3 Tuples

La fonctionnalité de tuples fournit une syntaxe concise pour regrouper plusieurs éléments de données dans une structure de données légère.
Exemple:

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

- les types de tuple prennent en charge les opérateurs d'égalité `==` et `!=`.
- les types de tuples sont des types valeur; les éléments de tuple sont des champs publics. Cela rend les types valeur mutable de tuples.

L'un des cas d'utilisation les plus courants de tuples est le type de retour de la méthode. Autrement dit, au lieu de définir des out en paramètres de méthode, on peut regrouper les résultats de méthode dans un type de retour de tuple.

On peut spécifier explicitement les noms des champs de tuple dans une expression d'initialisation de tuple ou dans la définition d'un type de tuple:
Exemple:

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");
```

C# prend en charge l'assignation entre les types tuple qui satisfont les deux conditions suivantes:

- les deux types de tuples ont le même nombre d'éléments
- pour chaque position de tuple, le type de l'élément de tuple de droite est le même que ou implicitement convertible en type de l'élément de tuple de gauche correspondant.

Les valeurs d'éléments tuples sont assignées à la suite de l'ordre des éléments du tuple.

9.4 Nullable

Un type valeur `Nullable T?` représente toutes les valeurs de son type valeur sous-jacent T et une valeur `null` supplémentaire. Par exemple, vous pouvez assigner l'une des trois valeurs suivantes à une `bool?` variable : true, `false` ou `null`. Un type valeur sous-jacent `T` ne peut pas être un type valeur Nullable lui-même.

Tout type valeur `Nullable` est une instance de la `System.Nullable<T>` structure générique. On peut faire référence à un type valeur `Nullable` avec un type sous-jacent `T` dans l'un des formulaires interchangeables suivants : `Nullable<T>` ou `T?`.

Exemple 1:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

Exemple 2:

```
public struct monNullable <T> where T:struct
{
    public monNullable (T value)
    {
        this.hasValue = true;
        this.value = value;
    }

    private bool hasValue;
    public bool HasValue
    {
        get
        {
            return hasValue;
        }
    }

    private T value;
    public T Value
    {
        get
        {
            if (!hasValue)
                throw new InvalidOperationException(" no value");
        return value;
        }
    }
}
```

10. Classes conteneurs génériques

Les types génériques suivants correspondent à des types de collections existants :

- `List<T>` est la classe générique qui correspond à ArrayList.
- `Dictionary<TKey,TValue>` et ConcurrentDictionary<TKey,TValue> sont les classes génériques qui correspondent à Hashtable.
- `Collection<T>` est la classe générique qui correspond à CollectionBase. Collection peut être utilisé comme classe de base, mais contrairement CollectionBase à, il n'est pas abstrait, ce qui le rend beaucoup plus facile à utiliser.
- `ReadOnlyCollection<T>` est la classe générique qui correspond à ReadOnlyCollectionBase. ReadOnlyCollection n'est pas abstraite et possède un constructeur qui facilite l'exposition d'un existant List en tant que collection en lecture seule.

- Les classes génériques, `Queue<T>`, `ConcurrentQueue<T>`, `ImmutableQueue<T>`, ImmutableArray, SortedList<TKey,TValue> et `ImmutableSortedSet<T>` correspondent aux classes non génériques respectives portant le même nom.
- LinkedList est une liste liée à usage général qui fournit des opérations d'insertion et de suppression O(1).
- `SortedDictionary<TKey,TValue>` est un dictionnaire trié avec des opérations d'insertion et d'extraction O(log n), ce qui en fait une alternative pratique à SortedList<TKey,TValue>.
- `KeyedCollection<TKey,TItem>` est un hybride entre une liste et un dictionnaire qui permet de stocker des objets qui contiennent leurs propres clés.
- `BlockingCollection<T>` implémente une classe de collection avec une fonctionnalité d'englobement et de blocage.
- `ConcurrentBag<T>` permet une insertion et une suppression rapides des éléments non triés.

10.1 List

Représente une liste fortement typée d'objets accessibles par index. Fournit des méthodes de recherche, de tri et de manipulation de listes.

Implémente:

- ICollection, IEnumerable, IList, IReadOnlyCollection, IReadOnlyList, ICollection, IEnumerable, IList

Propriété:

- Capacity
    - Obtient ou définit le nombre total des éléments que la structure de données interne peut contenir sans redimensionnement.
- Count
    - Obtient le nombre d'éléments contenus dans le List.
- Item[Int32]
    - Obtient ou définit l'élément au niveau de l'index spécifié.

Méthodes courantes:

- Add(T)
    - Ajoute un objet à la fin de la List.
- AddRange(IEnumerable)
    - Ajoute les éléments de la collection spécifiée à la fin de List.
- AsReadOnly()
    - Retourne un wrapper ReadOnlyCollection en lecture seule pour la collection actuelle.
- Clear()
    - Supprime tous les éléments de List.
- Contains(T)
    - Détermine si le List contient un élément.
- ConvertAll(Converter<T,TOutput>)
    - Convertit les éléments du List actuel dans un autre type et retourne une liste qui contient les éléments convertis.
- Exists(Predicate) (Hérité de Object)

    - Détermine si List contient des éléments qui correspondent aux conditions définies par le prédicat spécifié.
- Find(Predicate)
    - Recherche un élément qui correspond aux conditions définies par le prédicat spécifié et retourne la première occurrence dans le List entier.
- FindAll(Predicate)
    - Récupère tous les éléments qui correspondent aux conditions définies par le prédicat spécifié.
- ForEach(Action)
    - Exécute l'action spécifiée sur chaque élément de List.
- GetEnumerator()
    - Retourne un énumérateur qui itère au sein de List.
- GetHashCode()
    - Fait office de fonction de hachage par défaut.
- GetRange(Int32, Int32) (Hérité de Object)
    - Crée une copie superficielle d'une plage d'éléments de la source List.
- IndexOf(T) (Hérité de Object)
    - Recherche l'objet spécifié et retourne l'index de base zéro de la première occurrence trouvée dans l'ensemble du List.
- Insert(Int32, T)
    - Insère un élément dans la classe List au niveau de l'index spécifié.
- InsertRange(Int32, IEnumerable)
    - Insère les éléments d'une collection dans List au niveau de l'index spécifié.
- Remove(T) (Hérité de Object)
    - Supprime la première occurrence d’un objet spécifique de List.
- RemoveAll(Predicate)
    - Supprime tous les éléments qui correspondent aux conditions définies par le prédicat spécifié.
- Reverse()
    - Inverse l'ordre des éléments dans l'ensemble de List.
- Sort()
    - Trie les éléments dans l'ensemble de List à l'aide du comparateur par défaut.
- ToArray()
    - Copie les éléments de List dans un nouveau tableau.
- TrimExcess() (Hérité de Object)
    - Définit la capacité en fonction du nombre effectif d'éléments situés dans List, si ce nombre est inférieur à une valeur de seuil.

10.2 Queue

:::info
Représente une collection d’objets premier entré, premier sorti (FIFO).
:::

Trois opérations principales peuvent être effectuées sur un Queue et ses éléments :

- `Enqueue()` Ajoute un élément à la fin de Queue .
- `Dequeue()` supprime l’élément le plus ancien à partir du début de Queue .

- `Peek()` retourne l'élément le plus ancien qui se trouve au début de la `Queue<T>` mais ne le supprime pas.

La capacité d'une `Queue<T>` est le nombre d'éléments qu'elle peut contenir. À mesure que des éléments sont ajoutés à une Queue , la capacité est automatiquement augmentée en fonction des besoins en réallouant le tableau interne. La capacité peut être réduite en appelant TrimExcess.

`Queue<T>` accepte null comme valeur valide pour les types référence et autorise les éléments en double.

10.3 Stack

:::info
Représente une collection d'instances à taille variable de type dernier entré, premier sorti (LIFO) du même type spécifié.
:::

Trois opérations principales peuvent être effectuées sur un Stack et ses éléments :

- Push insère un élément en haut de Stack .
- Pop supprime un élément du haut de Stack .
- Peek retourne un élément qui se trouve en haut de, Stack mais ne le supprime pas de Stack .

La capacité d'un Stack est le nombre d'éléments que Stack peut contenir. À mesure que des éléments sont ajoutés à un Stack , la capacité est automatiquement augmentée en fonction des besoins en réallouant le tableau interne. La capacité peut être réduite en appelant TrimExcess.

Si `Count` est inférieur à la capacité de la pile, Push est une opération O(1). Si la capacité doit être augmentée pour s'adapter au nouvel élément, Push devient une opération O (n), où n est Count. Pop est une opération O(1).

Stack accepte `null` comme valeur valide pour les types référence et autorise les éléments en double.

10.4 Dictionary<Tkey , TValue>

:::info
Représente une collection de clés et de valeurs.

La classe générique `Dictionary<TKey,TValue>` fournit un mappage à partir d'un jeu de clés à un ensemble de valeurs.

Chaque ajout au dictionnaire se compose d'une valeur et de sa clé associée. La récupération d'une valeur à l'aide de sa clé est très rapide, proche de O (1), car la Dictionary<TKey,TValue> classe est implémentée en tant que table de hachage.
:::

Tant qu'un objet est utilisé comme clé dans le `Dictionary<TKey,TValue>` , il ne doit pas changer d'une manière qui affecte sa valeur de hachage. Chaque clé dans un `Dictionary<TKey,TValue>` doit être unique en fonction du comparateur d'égalité du dictionnaire. Une clé ne peut pas être null , mais une valeur peut être, si son type TValue est un type référence.

- .NET offre de nombreuses classes de dictionnaire, la principale est Dictionary<TKey, TValue>

Propriétés

- Comparer

    - Obtient le IEqualityComparer qui est utilisé pour déterminer l'égalité des clés pour le dictionnaire.
- Count
    - Obtient le nombre de paires clé/valeur contenues dans Dictionary<TKey,TValue>.
- Item[TKey]
    - Obtient ou définit la valeur associée à la clé spécifiée.
- Keys
    - Obtient une collection contenant les clés dans Dictionary<TKey,TValue>.
- Values
    - Obtient une collection contenant les valeurs dans Dictionary<TKey,TValue>.

Méthodes courantes:

- Add(TKey, TValue)
    - Ajoute la clé et la valeur spécifiées au dictionnaire.
- Clear()
    - Supprime toutes les clés et les valeurs de Dictionary<TKey,TValue>.
- ContainsKey(TKey)
    - Détermine si Dictionary<TKey,TValue> contient la clé spécifiée.
- ContainsValue(TValue)
    - Détermine si Dictionary<TKey,TValue> contient une valeur spécifique.
- GetHashCode()
    - Fait office de fonction de hachage par défaut.
- Remove(TKey)
    - Supprime de Dictionary<TKey,TValue> la valeur ayant la clé spécifiée.

10.5 HashSet<Tkey , TValue>

:::info
La classe HashSet fournit des opérations de définition de performances élevées. Un `set`, souvent appelé un ensemble, est une collection qui ne contient aucun élément en double, et dont les éléments ne sont pas dans un ordre particulier.
:::

C'est un peu comme un dictionnaire, mais sans les valeurs.

`HashSet<T>` fournit de nombreuses opérations ensemblistes mathématiques, telles que l’ajout de l’ensemble (unions) et la soustraction de la définition.

Le tableau suivant répertorie les opérations fournies et leurs équivalents mathématiques:

Opération HashSet	Équivalent mathématique
UnionWith	Ajout d'Union ou de jeu
IntersectWith	Ensembles
ExceptWith	Définir la soustraction
SymmetricExceptWith	Différence symétrique

En plus des *opérations de jeu* listées, la classe `HashSet<T>` fournit des méthodes permettant de déterminer l'égalité des ensembles, le chevauchement des jeux et si un ensemble est un sous-ensemble ou sur-ensemble d'un autre jeu.

LINQ fournit l'accès aux opérations `Distinct`, `Union`, `Intersect` et `Except` définies sur toute source de données qui implémente les interfaces `IEnumerable`, `IQueryable`.

`HashSet<T>` fournit une collection plus grande et plus robuste d'opérations que `Set`. Par exemple, des comparaisons telles que `IsSubsetOf` et `IsSupersetOf`.

11. `delegates`, expressions *λ*, events

11.1 `delegates`

Un délégué est un type qui représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers.

Les délégués sont utilisés pour passer des méthodes comme arguments à d'autres méthodes (passer des adresses de méthodes).

Ce sont des classes `type safe` qui définissent les types des paramètres et le type de retour.

Les gestionnaires d'événements sont tout simplement des méthodes appelées par le biais de délégués.

Exemple:

```
public delegate int PerformCalculation(int x, int y);
```

Vue d'ensemble:

- comparables aux pointeurs de fonction C++, sauf que les délégués sont totalement orientés objet, et contrairement aux pointeurs C++ vers les fonctions membres, ils encapsulent une instance d'objet et une méthode.
- permettent aux méthodes d'être transmises comme des paramètres.
- peuvent être utilisés pour définir des méthodes de rappel.
- peuvent être chaînés ; par exemple, plusieurs méthodes peuvent être appelées sur un seul événement.
- méthodes ne doivent pas nécessairement correspondre exactement au type délégué.
- les expressions lambda sont un moyen plus concis d'écrire des blocs de code inline. Les expressions lambda (dans certains contextes) sont compilées en types délégués.

Un `delegate` peut représenter plusieurs méthodes (multicast delegate), et sont en général des `Action<T>`:

- Ajout d'une méthode avec `+` et `+=`.
- Suppression d'une méthode avec `-` et `-=`.

11.2 Action et Func

Deux types de délégués génériques simplifient l'instanciation de délégués:

- Action délégué sans type de retour (0...16
- Func délégué avec un type de retour (0...16 arguments, puis type de retour)
  Exemple:

```
Func <in T1, in T2, in T3, in T4, out TResult
```

11.3 expressions *λ*

Pour créer une **expressions *λ***, il faut spécifier les paramètres d'entrée (le cas échéant) à gauche de l'opérateur lambda et une expression ou un bloc d'instructions de l'autre côté.

Toute **expressions *λ*** peut être convertie en type `delegate`.

Exemple 1:

```
public static void Each<T>(IEnumerable<T> items, Action<T> action)
{
    foreach (var item in items) action(item);
}

// utilisation avec lambda
DataUtil.Each(this.dataBase, data => hashSet.Add(data.GetColumn()));
```

Exemple 2, chaîne de lambda:

```
int GetPrimesCount (int min, int max)
{
    return Task.Run (
        () =>
        ParallelEnumerable
        .Range(min, max)
        .Count(n => Enumerable
               .Range(2, (int)Math.Sqrt(n)-1)
               .All(i => n%i > 0)) );
}
```

11.4 `events`

Les évènements sont basés sur les délégués et offrent un mécanisme de **publication-souscription** aux délégués.
Le mot clé `event` sert à définir un évènement.
Ils limitent l'interface d'un délégué pour que les abonnés ne puissent pas interférer les uns avec les autres.

- Les évènements utilisent des méthodes avec 2 paramètres, sans type de retour.

(**<object>** , **<TEventArgs>**)

- la référence de l'objet ayant émis l'évènement (p.ex. l'objet *Button* sur lequel on a clické)
- paramètre contenant des informations sur l'évènement (coordonnées d'un click souris)

Vue d'ensemble:

- Le publieur détermine quand un événement est déclenché; les abonnés déterminent l'action entreprise en réponse à l'événement.
- Un événement peut avoir plusieurs abonnés. Un abonné peut gérer plusieurs événements provenant de plusieurs publieurs.
- Les événements qui n'ont aucun abonné ne sont jamais déclenchés.
- Les événements sont généralement utilisés pour signaler des actions de l'utilisateur, comme les clics de bouton ou les sélections de menu dans les interfaces utilisateur graphiques.
- Quand un événement a plusieurs abonnés, les gestionnaires d'événements sont appelées de façon synchrone quand un événement est déclenché.
- Dans la bibliothèque de classes .NET, les événements sont basés sur le `EventHandler` délégué et la classe de base `EventArgs`.

Exemple:

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    // provide remaining implementation for the class
}
```

12. Énumérateur (yield)

Il est possible de créer des classes ou des méthodes utilisables à travers des boucles `foreach`.

Exemple:

```
class Counter
{
    public IEnumerable<int>(CountUp int start, int end)
    {
        int current = start;

        while(current <= end)
        {
            yield return current;
            current++;
        }
    }
}
```

13. Programmation asychrone (async, await)

.NET permet d'appeler n'importe quelle méthode de façon **asynchrone**. Pour ce faire, vous définissez un délégué avec la même signature que la méthode que vous souhaitez appeler. Le common language runtime définit automatiquement `BeginInvoke` les `EndInvoke` méthodes pour ce délégué, avec les signatures appropriées.

Définition de la méthode de test et du délégué asynchrone:

L'exemple de code suivant illustre la définition de `TestMethod()` et le délégué nommé `AsyncMethodCaller()` qui peut être utilisé pour appeler `TestMethod()` de manière asynchrone. Pour compiler les exemples de code, il est nécessaire d'inclure les définitions de `TestMethod()` et le délégué `AsyncMethodCaller()`.

Exemple:

```
using System;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncDemo
    {
        // The method to be executed asynchronously.
        public string TestMethod(int callDuration, out int threadId)
        {
            Console.WriteLine("Test method begins.");
            Thread.Sleep(callDuration);
            threadId = Thread.CurrentThread.ManagedThreadId;
            return String.Format("My call time was {0}.",
callDuration.ToString());
        }
    }
    // The delegate must have the same signature as the method
    // it will call asynchronously.
    public delegate string AsyncMethodCaller(int callDuration, out int
threadId);
}
```

Attente d'un appel asynchrone avec EndInvoke:

Le moyen le plus simple d'exécuter une méthode de manière asynchrone est de démarrer l'exécution de la méthode en appelant la méthode `BeginInvoke` du délégué, d'effectuer quelques tâches sur le thread principal, puis d'appeler la méthode `EndInvoke` du délégué. `EndInvoke` peut bloquer le thread appelant, car il ne retourne de pas résultat avant la fin de l'appel asynchrone. Cette technique est utile pour les opérations de fichier ou de réseau.

Exemple:

```
using System;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncMain
    {
        public static void Main()
        {
            // The asynchronous method puts the thread id here.
```

```
            int threadId;

            // Create an instance of the test class.
            AsyncDemo ad = new AsyncDemo();

            // Create the delegate.
            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            // Initiate the asychronous call.
            IAsyncResult result = caller.BeginInvoke(3000,
                out threadId, null, null);

            Thread.Sleep(0);
            Console.WriteLine("Main thread {0} does some work.",
                Thread.CurrentThread.ManagedThreadId);

            // Call EndInvoke to wait for the asynchronous call to complete,
            // and to retrieve the results.
            string returnValue = caller.EndInvoke(out threadId, result);

            Console.WriteLine("The call executed on thread {0}, with return value
\"{1}\".",
                threadId, returnValue);
        }
    }
}

/* This example produces output similar to the following:

Main thread 1 does some work.
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
 */
```