Mémo les différences des versions de C#

Et compréhension sur Framework .NET

Auteur : PIANETA Dimitri

V1 . 2021. Avril

Table des matières

Chapitre 1 : Comment fonctionne C# ?

Comme tout bon livre, je commencerai par un petit historique des versions de C#.

C# est un langage de programmation orienté objet, fortement type doté d'un typage statique dynamique et générique.

C# a été crée par Aders Hejlsberg et présenté officiellement en 2002 par Microsoft.

C# permet de développer des applications dans la plateforme Microsoft .NET.

C# est dérivé du C++ et influencé par JAVA : syntaxe, logique et concept assez proches.

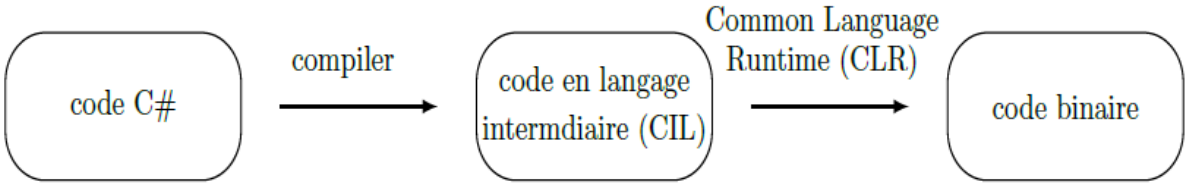
Permettant de développer des

- Application Web
- Application du bureau (Client lourd),
- Applications mobiles (sous Windows Phone),
- Services Web,
- Jeux,
- …

1) Comment ça fonctionne ?

On écrit un programme en C#. Le code C# sera transformé en langage intermédiaire (appelé CIL pour Common Intermediate Language).

Le code CIL sera compilé par la machine virtuelle (appelé CLR pour Common Language Runtime) pour avoir un code binaire.



Le *code CIL* est un code intermédiaire qu'on peut exécuter sur n'importe quelle machine Windows.

Le *code binaire* est adapté à la machine sur laquelle il tourne.

En plus, la machine virtuelle (CLR) dispose de :

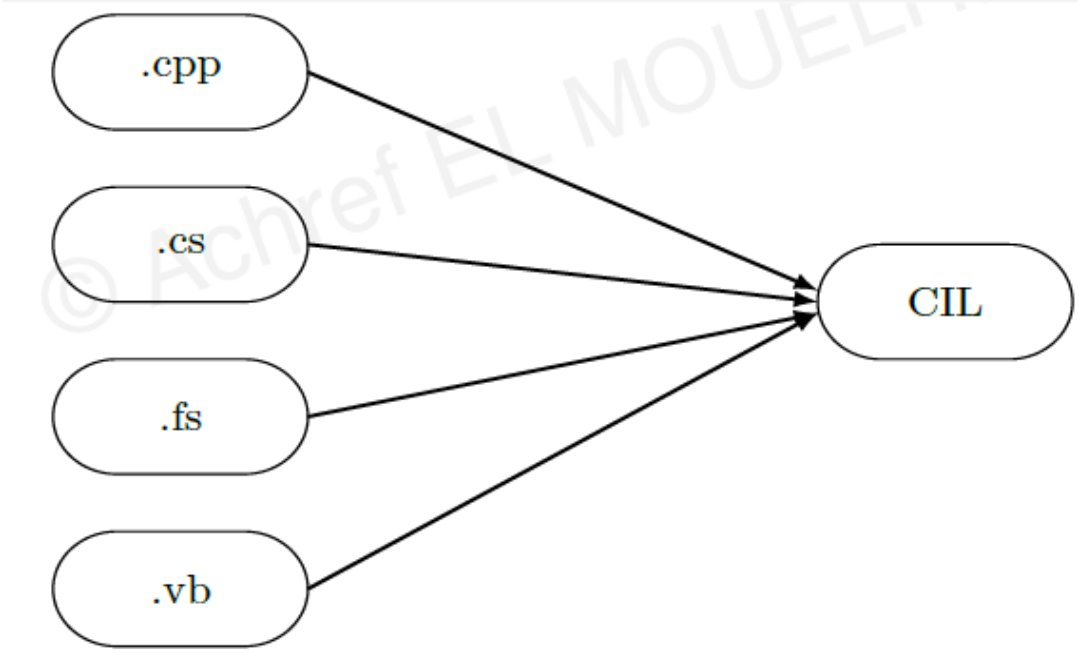
- JIT (Just In Time) : pour dubugger
- Garbage Collector : pour gérer la mémoire
- CTS (Common Type System) : fournir une bibliothèque contenant les types de données primitif

- CLS (Common Language Specification) : pour vérifier qu'un programme respecte les spécifications .NET.

2) Dans un framework .NET ?

On peut écrire un code C#, et aussi VB, C++, F#.

Tous ces langages seront compilés en code CIL.



A partir d'un programme C#, il est possible de créer :

- Soit un fichier .exe
- Soit une bibliothèque de classe sous la forme d'un fichier .dll

La différence vient que pour un .exe permet de lancer un programme et un .dll peut être par plusieurs programmes .exe.

Pour ces deux cas, il est appelé un assembly.

3) Les versions de C#

C# 1 (sortie en 2002) : classes, interfaces, opérateurs, structures, instructions...
C# 2 (sortie en 2005) : généricité, nullable, itérateurs, types partiels...
C# 3 (sortie en 2007) : **LINQ**...
C# 4 (sortie en 2010) : typage dynamique...
C# 5 (sortie en 2012) : programmation asynchrone...
C# 6 (sortie en 2015) : interpolation de chaîne de caractères, indexeur...
C# 7 (sortie en 2016) : tuples, out, ref...
C# 8 (sortie en 2019) : méthodes d'interface par défaut, ReadOnly...
C# 9 (sortie en 2020) : notion de Record...

Je vais aller plus en détails sur les changements entre les versions :

3.1 C# 1 :

Les principales fonctionnalités du langage C#1.0 étaient les suivantes qui sont toujours utilisés actuellement :

3.1.1 Classes :

Une classe est un type référence de mot clé *class*. Au moment de l'exécution, quand vous déclarez une variable de type référence, celle-ci contient la valeur Null tant que vous n'avez pas explicitement créé une instance de la classe à l'aide de l'opérateur *new* ou que vous ne lui avez pas assigné un objet existant d'un type compatible, comme indiqué par l'exemple suivant :

```
//Declaring an object of type MyClass.

MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.

MyClass mc2 = mc;
```

Quand l'objet est créé, une quantité de mémoire suffisante est allouée sur le tas managé de l'objet spécifié, et la variable contient uniquement une référence à l'emplacement de cet objet. Les types sur le tas managé entraînent une surcharge quand ils sont alloués et récupérés par la fonctionnalité de gestion automatique de la mémoire du CLR (appelée *garbage collection*).

La déclaration de classes :

Les classes sont déclarées à l'aide du mot clé *class* suivi d'un identificateur unique comme l'illustre l'exemple suivant :

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

Le mot clé `class` est précédé du niveau d'accès. Comme public est utilisé dans ce cas, n'importe qui peut créer des instances de cette classe. Le nom de la classe suit le mot clé `class`. Le nom de la classe doit être un nom d'identificateur C# valide. Le reste de la définition est le corps de la classe, où le comportement et les données sont définis. Les champs, propriétés, méthodes et événements d'une classe sont désignés collectivement par le terme « *membres de classe* ».

Création d'objets de classe :

Bien qu'ils soient parfois employés indifféremment, une classe et un objet sont deux choses différentes. Une classe définit un type d'objet, mais il ne s'agit pas d'un objet en soi. Un

objet, qui est une entité concrète basée sur une classe, est parfois désigné par le terme « instance de classe ».

Vous pouvez créer des objets en utilisant le mot clé new suivi du nom de la classe sur laquelle l'objet est basé, comme suit :

```
Customer object1 = new Customer();
```

Quand une instance d'une classe est créée, une référence à l'objet est repassée au programmeur. Dans l'exemple précédent, `object1` est une référence à un objet basé sur `Customer`. Cette référence fait référence au nouvel objet, mais elle ne contient pas ses données. En fait, vous pouvez créer une référence d'objet sans créer d'objet :

```
Customer object2;
```

Le site web de C# de Microsoft déconseille de créer des références d'objet comme celle-ci, sans référence à un objet, car toute tentative d'accès à un objet à l'aide d'une telle référence échoue au moment de l'exécution. Toutefois, une telle référence peut être faite pour faire référence à un objet, soit en créant un nouvel objet, soit en lui assignant un objet existant, tel que celui-ci :

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Héritage de classe :

Les classes prennent entièrement en charge l' *héritage* , caractéristique fondamentale de la programmation orientée objet. Lorsque vous créez une classe, vous pouvez hériter de toute autre classe qui n'est pas définie comme sealed, et d'autres classes peuvent hériter de votre classe et remplacer les méthodes virtuelles de la classe. En outre, vous pouvez implémenter une ou plusieurs interfaces.

L'héritage se fait par le biais d'une *dérivation* , ce qui signifie qu'une classe est déclarée à l'aide d'une *classe de base* dont elle hérite les données et le comportement. Pour spécifier une classe de base, ajoutez deux-points et le nom de la classe de base après le nom de la classe dérivée, comme suit :

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Un exemple finale :

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

3.1.2 Structures :

Un type de *structure* (ou *type struct*) est un type valeur qui peut encapsuler des données et des fonctionnalités associées. Le `struct` mot clé vous permet de définir un type de structure :

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

3.1.3 Interfaces :

Une interface contient des définitions pour un groupe de fonctionnalités connexes qu'une classe non abstraite ou un struct doit implémenter. Une interface peut définir des `static` méthodes, qui doivent avoir une implémentation.

Pour une interface, vous devez utiliser le mot clé interface.

Par exemple : dans un fichier d'interface en C#.

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

Le nom d'une interface doit être un nom d'identificateur C# valide. Par convention, les noms d'interface commencent par un `I` majuscule.

Toute classe ou tout struct qui implémentent l'interface IEquatable<T> doivent contenir une définition pour une méthode Equals qui correspond à la signature spécifiée par l'interface. Ainsi, vous pouvez compter sur une classe qui implémente `IEquatable<T>` pour contenir une méthode `Equals` avec laquelle une instance de la classe peut déterminer si elle est égale à une autre instance de la même classe.

La définition de `IEquatable<T>` ne fournit pas d'implémentation pour `Equals` . Une classe ou un struct peut implémenter plusieurs interfaces, mais une classe peut uniquement hériter d'une classe unique.

Ce qui nous donne le code de la classe implémenté de l'interface `IEquatable<T>` :

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==  (car.Make, car.Model, car.Year);
    }
}
```

Les propriétés et indexeurs d'une classe peuvent définir des accesseurs supplémentaires pour une propriété ou un indexeur qui est défini dans une interface. Par exemple, une interface peut déclarer une propriété qui a un accesseur get.

3.1.4 Événements :

Les événements sont, comme les délégués, un mécanisme de *liaison tardive*. En fait, les événements reposent sur la prise en charge linguistique des délégués. Ils permettent à un objet de signaler (à tous les composants du système intéressés), que quelque chose s'est produit. Tous les autres composants peuvent s'abonner à l'événement et être averti quand un événement est déclenché.

3.1.5 Propriétés :

Les propriétés auxquelles le code accède se comportent comme des champs. Toutefois, contrairement aux champs, les propriétés sont implémentées avec des accesseurs qui définissent quelles instructions sont exécutées au moment de l'accès à une propriété ou de son assignation.

3.1.6 Délégués :

Les délégués fournissent un mécanisme de *liaison tardive* dans .NET. La liaison tardive signifie que vous créez un algorithme où l'appelant fournit également au moins une méthode qui implémente une partie de l'algorithme.

Par exemple, prenez le tri d'une liste d'étoiles dans une application d'astronomie. Vous pouvez choisir de trier les étoiles d'après leur distance par rapport à la terre, d'après leur magnitude ou d'après leur luminosité perçue.

Dans tous ces cas-là, la méthode Sort() effectue essentiellement la même chose : elle organise les éléments dans la liste en se basant sur une comparaison. Le code qui compare deux étoiles est différent pour chacun des ordres de tri.

3.1.7 Opérateurs et expressions :

C# fournit un certain nombre d'opérateurs. Un grand nombre d'entre eux sont pris en charge par les types intégrés et vous permettent d'effectuer des opérations de base avec des valeurs de ces types. Ces opérateurs incluent les groupes suivants :

- **Opérateurs arithmétiques** qui effectuent des opérations arithmétiques avec des opérandes numériques
- **Opérateurs de comparaison** qui comparent des opérandes numériques
- **Opérateurs logiques booléens** qui effectuent des opérations logiques avec des `bool` opérandes
- **Opérateurs de bits et de décalage** qui effectuent des opérations de bits ou de décalage avec les opérandes des types intégraux
- **Opérateurs d'égalité** qui vérifient si leurs opérandes sont égaux ou non

En général, vous pouvez surcharger[1] ces opérateurs, autrement dit, spécifier le comportement de l'opérateur pour les opérandes d'un type défini par l'utilisateur.

3.1.8 Instructions

Les actions qu'un programme effectue sont exprimées dans les instructions. Les actions courantes incluent la déclaration de variables, l'assignation de valeurs, l'appel de méthodes, l'exécution de boucles dans les collections et la création de branches sur des blocs de code, selon une condition donnée. L'ordre dans lequel les instructions sont exécutées dans un programme est appelé le flux de contrôle ou le flux d'exécution. Le flux de contrôle peut varier chaque fois qu'un programme est exécuté, selon la manière dont le programme réagit à l'entrée qu'il reçoit au moment de l'exécution.

Une instruction peut comporter une seule ligne de code terminée par un point-virgule ou une série d'instructions d'une ligne dans un bloc. Un bloc d'instructions est placé entre des accolades ({}) et peut contenir des blocs imbriqués.

Par exemple :

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment
statement:
```

[1] Emploi d'un même nom pour désigner différentes constructions ; la surcharge est résolue statiquement par les compilateurs en fonction du contexte et de la signature.

```
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an
array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the
compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}.
Circumference = {2:N2}",
                                     counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types d'instructions :

Category	Mots clés C# / Remarques

Instructions de déclaration	Une instruction de déclaration introduit une nouvelle variable ou constante. Une déclaration de variable peut éventuellement assigner une valeur à la variable. Dans une déclaration de constante, l'assignation est obligatoire.
	Ex :
	`// Variable declaration statements.`
	`double area;`
	`double radius = 2;`
	
	`// Constant declaration statement.`
	`const double pi = 3.14159;`

Instructions d'expression	Les instructions d'expression qui calculent une valeur doivent stocker la valeur dans une variable.
	Ex :
	`// Expression statement (assignment).`
	`area = 3.14 * (radius * radius);`
	
	`// Error. Not  statement because no assignment:`
	`//circ * 2;`
	
	`// Expression statement (method invocation).`
	`System.Console.WriteLine();`
	
	`// Expression statement (new object creation).`
	`System.Collections.Generic.List<string> strings =`

	`new System.Collections.Generic.List<string>();`
Instructions de sélection	Les instructions de sélection permettent de créer des branches vers différentes sections de code, selon une ou plusieurs conditions spécifiées. Pour plus d'informations, voir les rubriques suivantes : • **if** • **else** • **switch** • **études**
Instructions d'itération	Les instructions d'itération permettent d'exécuter une boucle dans des collections telles que des tableaux, ou d'effectuer à plusieurs reprises le même jeu d'instructions jusqu'à ce qu'une condition spécifiée soit remplie. Pour plus d'informations, voir les rubriques suivantes : • **do** • **for** • **foreach** • **in** • **while**
Instructions de saut	Les instructions de saut transfèrent le contrôle vers une autre section de code. Pour plus d'informations, voir les rubriques suivantes : • **break** • **pouvoir** • **valeurs** • **goto** • **renvoi** • **yield**
Instructions de gestion des exceptions	Les instructions de gestion des exceptions permettent de récupérer normalement en cas de conditions exceptionnelles au moment de l'exécution. Pour plus d'informations, voir les rubriques suivantes : • **lever** • **try-catch** • **try-finally** • **try-catch-finally**
Activé et désactivé	Les instructions checked et unchecked vous permettent de spécifier si les opérations numériques sont autorisées à provoquer un dépassement de capacité lorsque le résultat est stocké dans une variable trop petite pour contenir la valeur résultante. Pour plus d'informations, consultez checked et unchecked.
Instruction `await`	Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression `await` dans la méthode async, il retourne à l'appelant, et la progression dans la méthode est interrompue jusqu'à ce que la tâche attendue soit terminée. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode. Pour obtenir un exemple simple, consultez la section « Méthodes async » de Méthodes.
Instruction `yield return`	Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution redémarre à partir de cet emplacement au prochain appel de l'itérateur.
Instruction `fixed`	L'instruction fixed empêche le récupérateur de mémoire de déplacer une variable mobile. Pour plus d'informations, consultez fixed.
Instruction `lock`	L'instruction lock permet de limiter l'accès aux blocs de code à un seul thread à la fois. Pour plus d'informations, consultez lock.

Instructions étiquetées	Vous pouvez donner une étiquette à une instruction, puis utiliser le mot clé goto pour sauter jusqu'à l'instruction étiquetée.

Instruction vide	L'instruction vide se compose seulement d'un point-virgule. Elle ne fait rien et peut être utilisée à un emplacement où une instruction est requise alors qu'aucune action ne doit être effectuée.
	Ex :
	
	`void ProcessMessages()`
	`{`
	`  while (ProcessMessage())`
	`  ; // Statement needed here.`
	`}`
	
	`void F()`
	`{`
	`  //...`
	`  if (done) goto exit;`
	`//...`
	`exit:`
	`  ; // Statement needed here.`
	`}`

Instructions incorporées :

Certaines instructions, y compris do, while, for et foreach, sont toujours suivies d'une instruction incorporée. Cette instruction incorporée peut être une seule instruction ou plusieurs instructions placées entre des accolades ({}) dans un bloc d'instructions. Même les instructions incorporées d'une seule ligne peuvent être placées entre des accolades ({}), comme illustré dans l'exemple suivant :

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

Une instruction incorporée qui n'est pas placée entre des accolades ({}) ne peut pas être une instruction de déclaration ni une instruction étiquetée. Ceci est illustré dans l'exemple suivant :

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Placez l'instruction incorporée dans un bloc pour corriger l'erreur :

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Blocs d'instructions incorporées :

Les blocs d'instructions peuvent être imbriqués, comme illustré dans le code suivant :

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

3.1.9 Attributs

Les attributs fournissent une méthode puissante permettant d'associer des métadonnées ou des informations déclaratives avec du code (assemblys, types, méthodes, propriétés, etc.). Une fois associé à une entité de programme, l'attribut peut être interrogé à l'exécution à l'aide d'une technique appelée *réflexion.*

3.2 C#1. 2 :

La version C# 1,2 est fournie avec Visual Studio .NET 2003. Cette version contenait quelques améliorations mineures du langage. La principale est que, à compter de cette version, le code était généré dans une boucle `foreach` (appelée Dispose) sur un IEnumerator quand ce IEnumerator implémentait IDisposable.

3.3 C#2.0 :

Les choses commencent alors à devenir intéressantes. Examinons certaines fonctionnalités majeures de C# 2.0, sorti en 2005, en même temps que Visual Studio 2005 :

3.3.1 Génériques :

Les génériques introduisent le concept de paramètres de type dans .NET, ce qui permet de concevoir des classes et des méthodes qui diffèrent la spécification d'un ou plusieurs types jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code client.

Par exemple, en utilisant un paramètre de type générique `T` , vous pouvez écrire une seule classe qui peut être utilisée par un autre code client sans impliquer le coût ou le risque des casts ou des opérations de boxing du runtime, comme illustré ici :

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Les classes et les méthodes génériques combinent la réutilisabilité, la cohérence des types et l'efficacité d'une façon que leurs équivalents non génériques ne peuvent pas. Les génériques sont plus fréquemment utilisés dans des collections et des méthodes qui agissent sur eux. L' System.Collections.Generic espace de noms contient plusieurs classes de collection génériques. Les collections non génériques, telles que, ne ArrayList sont pas recommandées et sont conservées à des fins de compatibilité.

Bien sûr, vous pouvez également créer des types et des méthodes génériques personnalisés pour fournir des solutions et des modèles de conception généralisés qui soient efficaces et de type sécurisé. L'exemple de code suivant montre une classe de liste liée générique simple, à des fins de démonstration. (Dans la plupart des cas, vous devez utiliser la List<T> classe fournie par .net au lieu de créer la vôtre.) Le paramètre `T` de type est utilisé dans plusieurs emplacements où un type concret est normalement utilisé pour indiquer le type de l'élément stocké dans la liste. Il est utilisé de la façon suivante :

- Comme le type d'un paramètre de méthode dans la méthode`AddHead`.
- Comme le type de retour de la propriété `Data` de la classe imbriquée `Node`.
- Comme le type de membre privé `data` de la classe imbriquée.

Notez que `T` est disponible pour la classe imbriquée `Node` . Quand `GenericList<T>` est instancié avec un type concret, par exemple comme un `GenericList<int>`, chaque occurrence de `T` est remplacée par `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
```

```
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

L'exemple de code suivant montre comment le code client utilise la classe générique `GenericList<T>` pour créer une liste d'entiers. En changeant l'argument de type, vous pouvez facilement modifier le code suivant pour créer des listes de chaînes ou tout autre type personnalisé :

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

3.3.2 Types partiels :

Il est possible de fractionner la définition d'une classe, d'un struct, d'une interface ou d'une méthode entre plusieurs fichiers sources. Chaque fichier source contient une section de la définition de méthode ou de type, et toutes les parties sont combinées au moment où l'application est compilée.

Classes partielles :

Il peut être utile de fractionner une définition de classe dans les situations suivantes :

- Dans des projets volumineux, le fractionnement d'une classe entre plusieurs fichiers distincts permet à plusieurs programmeurs d'y travailler simultanément.
- Quand vous utilisez une source générée automatiquement, vous pouvez ajouter du code à la classe sans avoir à recréer le fichier source. Visual Studio suit cette approche pour créer des formulaires Windows Forms, du code wrapper de service web, etc. Vous pouvez écrire du code qui utilise ces classes sans avoir à modifier le fichier créé par Visual Studio.
- Pour fractionner une définition de classe, utilisez le modificateur de mot clé partial, comme ci-dessous :

```
public  partial  class  Employee

{

    public void DoWork()

    {

    }

}

public  partial  class  Employee

{

    public void GoToLunch()
```

```
    {

    }

}
```

Le mot clé `partial` indique que d'autres parties de la classe, du struct ou de l'interface peuvent être définies dans l'espace de noms. Toutes les parties doivent utiliser le mot clé `partial`. Toutes les parties doivent être disponibles à la compilation pour former le type final. Toutes les parties doivent avoir la même accessibilité : `public`, `private`, etc.

Restrictions :

Il y a plusieurs règles à respecter quand vous utilisez des définitions de classe partielle :

- Toutes les définitions de type partiel conçues comme des parties du même type doivent être modifiées avec `partial`. Par exemple, les déclarations de classe suivantes génèrent une erreur :

```
public partial class A { }
//public class A { }  // Error, must also be marked partia
```

- Le modificateur `partial` peut uniquement être placé juste avant les mots clés `class`, `struct` ou `interface`.

- Les types partiels imbriqués sont autorisés dans les définitions de type partiel, comme illustré dans l'exemple suivant :

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- Toutes les définitions de type partiel conçues comme des parties du même type doivent être définies dans le même assembly et dans le même module (fichier .exe ou .dll). Les définitions partielles ne peuvent pas être fractionnées entre plusieurs modules.

- Le nom de classe et les paramètres de type générique doivent correspondre dans toutes les définitions de type partiel. Les types génériques peuvent être partiels. Chaque déclaration partielle doit utiliser les mêmes noms de paramètres, dans le même ordre.

- Les mots clés suivants sont facultatifs dans une définition de type partiel. S'ils sont utilisés dans une définition de type partiel, ils ne doivent pas être en conflit avec les mots clés spécifiés dans une autre définition partielle pour le même type :

- public
- private
- Protected
- internal
- abstraction
- sealed
- classe de base
- modificateur new (parties imbriquées)
- contraintes génériques

3.3.3 Méthodes anonymes :

L'opérateur `delegate` crée une méthode anonyme qui peut être convertie en un type délégué :

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

À partir C# 3, les expressions lambda offrent un moyen plus concis et plus expressif de créer une fonction anonyme. Utilisez => opérateur pour construire une expression lambda :

C#

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4));  // output: 7
```

Lorsque vous utilisez l'opérateur `delegate`, vous pouvez omettre la liste de paramètres. Dans ce cas, la méthode anonyme créée peut être convertie en un type délégué avec n'importe quelle liste de paramètres, comme le montre l'exemple suivant :

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is
world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

3.3.4 Types valeur 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. Vous pouvez faire référence à un type valeur Nullable avec un type sous-jacent `T` dans l'un des formulaires interchangeables suivants : `Nullable<T>` ou `T?` .

En général, vous utilisez un type valeur Nullable lorsque vous devez représenter la valeur indéfinie d'un type valeur sous-jacent. Par exemple, une variable booléenne, ou `bool` , ne peut être que `true` ou `false` . Toutefois, dans certaines applications, une valeur de variable peut être indéfinie ou manquante. Par exemple, un champ de base de données peut contenir `true` ou `false` , ou il ne peut contenir aucune valeur, autrement dit, `NULL` . Vous pouvez utiliser le `bool?` type dans ce scénario.

Déclaration et affectation :

```
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];
```

Examen d'une instance d'un type valeur Nullable :

À compter de C# 7,0, vous pouvez utiliser l' **`is`** opérateur avec un modèle de type pour examiner à la fois une instance d'un type de valeur Nullable pour `null` et récupérer une valeur d'un type sous-jacent :

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

Vous pouvez toujours utiliser les propriétés en lecture seule suivantes pour examiner et obtenir la valeur d'une variable de type valeur Nullable :

- Nullable<T>.HasValue indique si une instance d'un type valeur Nullable a une valeur de son type sous-jacent.
- Nullable<T>.Value obtient la valeur d'un type sous-jacent si HasValue est `true`. Si HasValue est `false`, la propriété Value lève une InvalidOperationException.

L'exemple suivant utilise la `HasValue` propriété pour déterminer si la variable contient une valeur avant de l'afficher :

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

Vous pouvez également comparer une variable d'un type valeur Nullable avec `null` au lieu d'utiliser la `HasValue` propriété, comme le montre l'exemple suivant

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Conversion d'un type valeur Nullable en un type sous-jacent :

Si vous souhaitez assigner une valeur d'un type valeur Nullable à une variable de type valeur n'acceptant pas les valeurs NULL, vous devrez peut-être spécifier la valeur à assigner à la place de `null` . Utilisez l' opérateur ?? de fusion Null pour effectuer cette opération (vous pouvez également utiliser la Nullable<T>.GetValueOrDefault(T) méthode dans le même but) :

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

Si vous souhaitez utiliser la valeur par défaut du type valeur sous-jacent à la place de `null` , utilisez la Nullable<T>.GetValueOrDefault() méthode.

Vous pouvez également effectuer un cast explicite d'un type valeur Nullable vers un type non Nullable, comme le montre l'exemple suivant :

```
int? n = null;

//int m1 = n;     // Doesn't compile
```

```
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

Au moment de l'exécution, si la valeur d'un type valeur Nullable est `null` , le cast explicite lève une InvalidOperationException .

Un type valeur n'acceptant pas les valeurs null `T` est implicitement convertible en type valeur Nullable correspondant `T?` .

Opérateurs levés

Les opérateurs unaires et binaires prédéfinis ou tous les opérateurs surchargés pris en charge par un type valeur `T` sont également pris en charge par le type valeur Nullable correspondant `T?` . Ces opérateurs, également appelés *opérateurs levés*, produisent `null` si l'un des opérandes ou les deux sont `null` ; sinon, l'opérateur utilise les valeurs contenues de ses opérandes pour calculer le résultat. Par exemple :

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

Boxing et unboxing

Une instance d'un type valeur Nullable `T?` est convertie comme suit :

- Si HasValue retourne `false`, la référence null est générée.
- Si HasValue retourne `true` , la valeur correspondante du type valeur sous-jacent `T` est boxed, et non l'instance de Nullable<T> .

Vous pouvez convertir une valeur boxed d'un type valeur `T` en type valeur Nullable correspondant `T?` , comme le montre l'exemple suivant :

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

Comment identifier un type valeur Nullable :

L'exemple suivant montre comment déterminer si une System.Type instance représente un type valeur Nullable construit, autrement dit, le System.Nullable<T> type avec un paramètre de type spécifié `T` :

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non
nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-
nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

3.3.5 Iterators :

Un *itérateur* peut être utilisé pour parcourir des collections, comme des listes et des tableaux.

Une méthode d'itérateur ou un accesseur `get` effectue une itération personnalisée sur une collection. Une méthode d'itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution est redémarrée à partir de cet emplacement lors de l'appel suivant de la fonction d'itérateur.

Vous consommez un itérateur à partir du code client en utilisant une instruction foreach ou une requête LINQ.

Dans l'exemple suivant, la première itération de la boucle `foreach` fait que l'exécution continue dans la méthode d'itérateur `SomeNumbers`, jusqu'à ce que la première instruction `yield return` soit atteinte. Cette itération retourne la valeur 3, et l'emplacement actif dans la méthode d'itérateur est mémorisé. Dans l'itération suivante de la boucle, l'exécution de la méthode d'itérateur continue là où elle s'était arrêtée, et s'arrête de nouveau quand elle atteint une instruction `yield return`. Cette itération retourne la valeur 5, et l'emplacement actif dans la méthode d'itérateur est mémorisé. La boucle se termine quand la fin de la méthode d'itérateur est atteinte.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

Le type de retour d'une méthode d'itérateur ou de l'accesseur `get` peut être IEnumerable, IEnumerable<T>, IEnumerator ou IEnumerator<T>.

Utilisez une instruction `yield break` pour terminer l'itération

Itérateur simple

L'exemple suivant comprend une seule instruction `yield return` qui se trouve dans une boucle for. Dans `Main`, chaque itération du corps d'instruction `foreach` crée un appel à la fonction d'itérateur, qui poursuit avec l'instruction `yield return` suivante.

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Création d'une classe de collection :

Dans l'exemple suivant, la classe `DaysOfTheWeek` implémente l'interface IEnumerable, ce qui nécessite la méthode GetEnumerator. Le compilateur appelle implicitement la méthode `GetEnumerator`, qui retourne un IEnumerator.

La méthode `GetEnumerator` retourne chaque chaîne une à la fois en utilisant l'instruction `yield return`.

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
```

```
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

3.3.6 Covariance et contravariance :

En C#, la covariance et la contravariance permettent la conversion de références implicite pour les types tableau, les types délégués et les arguments de type générique. La covariance conserve la compatibilité d'assignation et la contravariance l'inverse.

```
/ Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less
derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

La covariance pour les tableaux permet la conversion implicite d'un tableau d'un type plus dérivé en un tableau d'un type moins dérivé. Cette opération n'est cependant pas sécurisée au niveau des types, comme le montre l'exemple de code suivant.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

La prise en charge de la covariance et la contravariance pour les groupes de méthodes permet la correspondance des signatures de méthode avec des types délégués. Ceci vous permet d'affecter aux délégués non seulement les méthodes ayant des signatures correspondantes, mais aussi des méthodes qui retournent des types plus dérivés (covariance) ou qui acceptent des paramètres ayant des types moins dérivés (contravariance) que ceux spécifiés par le type délégué.

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

3.3.7 Autres changements :

D'autres fonctionnalités de C# 2.0 ajoutaient des capacités aux fonctionnalités existantes :

- Accessibilité distincte des accesseurs Get/Set
- Conversions de groupes de méthodes (délégués)
- Classes statiques
- Inférence de délégué

Alors que C# avait démarré en tant que langage orienté objet générique, C# version 2.0 a rapidement changé tout ça. En effet, maintenant que le langage existait, il s'agissait de s'attaquer à certains problèmes majeurs que rencontraient les développeurs. Et de s'y attaquer de manière significative.

Avec les génériques, les types et les méthodes peuvent fonctionner sur un type arbitraire tout en assurant quand même la cohérence des types. À titre d'exemple, un List<T> vous permet d'avoir `List<string>` ou `List<int>` et d'effectuer des opérations qui maintiennent la cohérence des types sur des chaînes ou entiers alors que vous itérez en leur sein. L'utilisation de génériques est préférable à la création d'un `ListInt` type dérivant de `ArrayList` ou effectuant un cast à partir de `Object` pour chaque opération.

C# version 2.0 a introduit les itérateurs. En bref, les itérateurs permettent d'examiner tous les éléments dans un `List` (ou d'autres types énumérables) avec une boucle `foreach`. Le fait de disposer d'itérateurs comme composants de premier ordre du langage en a considérablement amélioré la lisibilité et la capacité de raisonnement des utilisateurs vis-à-vis du code.

3.4 C#3.0 :

C# version 3.0 est apparu fin 2007, en même temps que Visual Studio 2008, même si l'éventail complet des fonctionnalités du langage n'a réellement voir le jour qu'avec le .NET Framework version 3.5. Cette version a insufflé un changement majeur dans l'évolution de C#. Elle a imposé C# en tant que langage de programmation vraiment formidable. Examinons certaines fonctionnalités importantes dans cette version :

- Propriétés implémentées automatiquement
- Types anonymes
- Expressions de requête
- Expressions lambda
- Arborescences d'expressions
- Méthodes d'extension
- Variables locales implicitement typées
- Méthodes partielles
- Initialiseurs d'objets et de collections

Rétrospectivement, nombre de ces fonctionnalités semblent à la fois inéluctables et inséparables. Elles s'assemblent de façon stratégique. L'expression de requête, également appelée LINQ (Language-Integrated Query), était globalement considérée comme la fonctionnalité remarquable de cette version de C#.

Un point de vue plus nuancé s'intéresse aux arborescences d'expressions, aux expressions lambda et aux types anonymes sur lesquels se construit LINQ. Mais, dans les deux cas, C# 3.0 présentait un concept révolutionnaire. C# 3.0 jetait les bases qui allaient permettre à C# de devenir un langage fonctionnel/orienté objet hybride.

Plus précisément, il devint alors possible d'écrire des requêtes déclaratives de style SQL pour effectuer des opérations sur des collections, entre autres choses. Au lieu d'écrire une boucle `for` pour calculer la moyenne d'une liste d'entiers, vous pouviez alors simplement utiliser `list.Average()`. La combinaison des expressions de requête et des méthodes d'extension a permis à de telles listes d'entiers de paraître beaucoup plus intelligentes.

Il a fallu du temps aux utilisateurs pour comprendre et intégrer ce concept, mais ils y sont progressivement parvenus. Et aujourd'hui, bien des années plus tard, le code est beaucoup plus concis, simple et fonctionnel.

3.5 C#4.0 :

La version 4,0 de C#, publiée avec Visual Studio 2010, aurait eu un temps difficile à atteindre l'État révolutionnaire de la version 3,0. Avec la version 3.0, le langage C# est bel et bien sorti de l'ombre de Java pour être propulsé sur le devant de la scène. Il allait même rapidement devenir élégant.

La version suivante introduisit de nouvelles fonctionnalités intéressantes :

- Liaison dynamique
- Arguments nommés/facultatifs

- Covariance et contravariance génériques
- Types interop incorporés

Les types interop incorporés ont permis d'atténuer une difficulté de déploiement. La covariance et la contravariance génériques vous donnent plus de contrôle sur l'utilisation des génériques, mais elles sont un peu académiques et probablement plus appréciées des auteurs de frameworks et de bibliothèques. Les paramètres nommés et facultatifs vous permettent d'éliminer les nombreuses surcharges de méthode et s'avèrent plus pratiques. Mais aucune de ces fonctionnalités ne représente un vrai changement de paradigme.

La fonctionnalité majeure était plutôt l'introduction du mot clé `dynamic`. Le mot clé `dynamic` introduisit dans C# version 4.0 la possibilité de remplacer le compilateur lors de la saisie au moment de la compilation. En utilisant le mot clé dynamic, vous pouvez créer des constructions semblables aux langages dynamiquement typés comme JavaScript. Vous pouvez créer un `dynamic x = "a string"`, puis y ajouter six, et ainsi laisser le runtime se débrouiller avec ce qui doit se produire par la suite.

La liaison dynamique offre la possibilité de commettre des erreurs, mais également de contrôler davantage le langage.

3.6 C#5.0 :

C# version 5,0, fourni avec Visual Studio 2012, était une version ciblée du langage. Presque tous les efforts déployés pour cette version portaient sur un autre concept révolutionnaire du langage : le modèle `async` et `await` pour la programmation asynchrone. Voici la liste des fonctionnalités principales :

- Membres asynchrones
- Attributs d'informations de l'appelant

3.7 C#6.0 :

Avec les versions 3.0 et 5.0, C# avait ajouté d'importantes nouvelles fonctionnalités à un langage orienté objet. Avec la version 6,0, publiée avec Visual Studio 2015, il ne s'agissait pas d'une fonctionnalité de déploiement dominant et de libérer à la place de nombreuses fonctionnalités plus petites qui rendaient la programmation C# plus productive. En voici quelques-unes :

- Importations statiques
- Filtres d'exceptions
- Initialiseurs de propriétés automatiques
- Membres expression-bodied
- Propagateur Null
- Interpolation de chaîne
- opérateur nameof

Quelques autres nouvelles fonctions :

- Initialiseurs d'index

- Await dans des blocs catch/finally
- Valeurs par défaut pour les propriétés d'accesseur Get

Chacune de ces fonctionnalités est intéressante individuellement. Mais si vous les examinez dans leur ensemble, un modèle intéressant se dégage. Dans cette version, C# a éliminé le texte réutilisable du langage pour rendre le code plus laconique et plus lisible. Ainsi, pour les amateurs de code propre et simple, cette version du langage était une énorme victoire.

3.8 C#7.0 :

C# version 7,0 a été publié avec Visual Studio 2017. Cette version propose des évolutions intéressantes dans l'esprit de C# 6.0, mais sans le compilateur en tant que service. Voici quelques-unes des nouvelles fonctionnalités :

- Variables out
- Tuples et déconstruction
- Critères spéciaux
- Fonctions locales
- Membres expression-bodied étendus
- Variables locales et retours ref

Autres fonctionnalités disponibles :

- Éléments ignorés
- Littéraux binaires et séparateurs numériques
- Expressions throw

Toutes ces fonctionnalités offrent de nouvelles capacités appréciables aux développeurs ainsi que la possibilité d'écrire du code encore plus propre. Il s'agit notamment de condenser la déclaration des variables à utiliser avec le mot clé `out` et en autorisant plusieurs valeurs de retour par le biais d'un tuple.

3.8 C#7.1 :
C# a démarré la publication de *versions point* avec c# 7,1. Cette version a ajouté l'élément de configuration de la sélection de version de langage , trois nouvelles fonctionnalités de langage et un nouveau comportement de compilateur.

Les nouvelles fonctionnalités de langage de cette version sont :

- `async``Main`méthode
    - Le point d'entrée pour une application peut avoir le modificateur `async`.
- `default` expressions littérales
    - Vous pouvez utiliser des expressions littérales default dans les expressions de valeur par défaut quand le type cible peut être inféré.
- Noms des éléments de tuple inférés
    - Les noms des éléments de tuple peuvent être inférés dans de nombreux cas à partir de l'initialisation du tuple.

- Critères spéciaux sur les paramètres de type générique

    - Il est possible d'utiliser des expressions de critères spéciaux sur les variables dont le type est un paramètre de type générique.

Enfin, le compilateur a deux options, `-refout` et `-refonly`, qui contrôlent la génération d'assemblys de références.

3.9 C#7.2 :

C# 7,2 a ajouté plusieurs fonctionnalités de langage de petite taille :

- Techniques d'écriture de code safe et efficace

    - Une combinaison des améliorations de la syntaxe qui permettent d'utiliser les types valeur avec la sémantique de référence.
- Arguments nommés non placés en position de fin

    - Les arguments nommés peuvent être suivis par des arguments de position.
- Traits de soulignement de début dans les littéraux numériques

    - Les littéraux numériques peuvent maintenant comporter des traits de soulignement de début avant tout chiffre affiché.
- `private protected` modificateur d'accès

    - Le modificateur d'accès `private protected` active l'accès pour les classes dérivées dans le même assembly.
- Expressions conditionnelles `ref`

    - Le résultat d'une expression conditionnelle (`?:`) peut maintenant être une référence.

3.10 C#7.3 :

Il existe deux thèmes principaux pour la version C# 7.3. Un thème fournit des fonctionnalités permettant au code sécurisé d'être aussi performant que le code non sécurisé. Le second thème fournit des améliorations incrémentielles aux fonctionnalités existantes. En outre, de nouvelles options de compilateur ont été ajoutées à cette version.

Les nouvelles fonctionnalités suivantes prennent en charge le thème de meilleures performances pour le code sécurisé :

- Vous pouvez accéder à des champs fixes sans épinglage.
- Vous pouvez réassigner des `ref` variables locales.
- Vous pouvez utiliser des initialiseurs sur des `stackalloc` tableaux.
- Vous pouvez utiliser des `fixed` instructions avec n'importe quel type prenant en charge un modèle.
- Vous pouvez utiliser des contraintes génériques supplémentaires.

Les améliorations suivantes ont été apportées aux fonctionnalités existantes :

- Vous pouvez tester == et `!=` avec des types de tuple.
- Vous pouvez utiliser des variables d'expression dans d'autres emplacements.

- Vous pouvez joindre des attributs à un champ de stockage de propriétés implémentées automatiquement.
- La résolution de la méthode lorsque des arguments diffèrent de `in` a été améliorée.
- La résolution de la surcharge comporte maintenant moins de cas ambigus.

Les nouvelles options du compilateur sont les suivantes :

- `-publicsign` pour activer la signature d'assemblys Open Source Software (OSS).
- `-pathmap` pour fournir un mappage des répertoires sources.

3.11 C#8.0 :

C# 8,0 est la première version majeure de C# qui cible spécifiquement .NET Core. Certaines fonctionnalités s'appuient sur les nouvelles fonctionnalités CLR, d'autres sur les types de bibliothèque ajoutés uniquement dans .NET Core. C# 8,0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Membres ReadOnly
- Méthodes d'interface par défaut
- Améliorations des critères spéciaux:
  - Expressions de commutateur
  - Modèles de propriétés
  - Modèles de tuples
  - Modèles positionnels
- Utilisation des déclarations
- Fonctions locales statiques
- Structs ref jetables
- Types références Nullables
- Flux asynchrones
- Index et plages
- Assignation de fusion Null
- Types construits non managés
- Stackalloc dans les expressions imbriquées
- Amélioration des chaînes textuelles interpolées

Les membres d'interface par défaut nécessitent des améliorations dans le CLR. Ces fonctionnalités ont été ajoutées dans le CLR pour .NET Core 3,0. Les plages et les index, et les flux asynchrones requièrent de nouveaux types dans les bibliothèques .NET Core 3,0. Les types de référence Nullable, bien qu'implémentés dans le compilateur, sont bien plus utiles lorsque les bibliothèques sont annotées pour fournir des informations sémantiques sur l'État null des arguments et les valeurs de retour. Ces annotations sont ajoutées dans les bibliothèques .NET Core.

4) Les nouveautés des versions de la 7.0 à 9.0

4.1 De C# 7.0 à 7.3

C# 7,0 à C# 7,3 a apporté un certain nombre de fonctionnalités et des améliorations incrémentielles à votre expérience de développement avec C#. Cet article fournit une vue d'ensemble des nouvelles fonctionnalités de langage et des options du compilateur. Les descriptions décrivent le comportement de C# 7,3, qui est la version la plus récente prise en charge pour les applications .NET Framework.

L'élément de configuration de sélection de la version de langage a été ajouté avec C# 7,1, ce qui vous permet de spécifier la version du langage du compilateur dans votre fichier projet.

C# 7.0-7.3 ajoute ces fonctionnalités et thèmes au langage C# :

- Tuples et éléments ignorés
  - Vous pouvez créer des types légers et sans nom qui contiennent plusieurs champs publics. Les compilateurs et les outils de l'IDE comprennent la sémantique de ces types.
  - Les éléments ignorés sont les variables temporaires en écriture seule utilisées dans les attributions quand vous ne vous souciez pas de la valeur assignée. Ils s'avèrent utiles lors de la déconstruction de tuples et de types définis par l'utilisateur, ainsi que lors de l'appel de méthodes avec des paramètres `out`.
- Critères spéciaux
  - Vous pouvez créer une logique de branchement basée sur des types arbitraires et les valeurs des membres de ces types.
- `async``Main`méthode
  - Le point d'entrée pour une application peut avoir le modificateur `async`.
- Fonctions locales
  - Vous pouvez imbriquer des fonctions dans d'autres fonctions afin de limiter leur portée et leur visibilité.
- Autres membres expression-bodied
  - La liste des membres pouvant être créés à l'aide d'expressions s'est allongée.
- `throw` Manifestations
  - Vous pouvez lever des exceptions dans les constructions de code qui n'étaient pas autorisées auparavant, car `throw` était une instruction.
- default expressions littérales
  - Vous pouvez utiliser des expressions littérales default dans les expressions de valeur par défaut quand le type cible peut être inféré.
- Améliorations de la syntaxe littérale numérique
  - De nouveaux jetons améliorent la lisibilité des constantes numériques.
- out variables
  - Vous pouvez déclarer des valeurs `out` inline comme arguments de la méthode dans laquelle elles sont utilisées.
- Arguments nommés non placés en position de fin
  - Les arguments nommés peuvent être suivis par des arguments de position.
- private protected modificateur d'accès
  - Le modificateur d'accès `private protected` active l'accès pour les classes dérivées dans le même assembly.
- Résolution de surcharge améliorée
  - Nouvelles règles pour résoudre l'ambiguïté de résolution de surcharge.
- Techniques d'écriture de code safe et efficace
  - Une combinaison des améliorations de la syntaxe qui permettent d'utiliser les types valeur avec la sémantique de référence.

Enfin, le compilateur a de nouvelles options :

- `-refout` et `-refonly` qui contrôlent la génération de l' assembly de référence.
- `-publicsign` pour activer la signature d'assemblys Open Source Software (OSS).
- `-pathmap` pour fournir un mappage des répertoires sources.

4.1.1 Tuples et éléments ignorés

C# fournit une syntaxe complète pour les classes et les structs, utilisée pour expliquer l'intention de votre conception. Cependant, cette syntaxe complète nécessite parfois un travail supplémentaire avec peu d'avantages. Vous pouvez souvent écrire des méthodes qui nécessitent une structure simple contenant plusieurs éléments de données. Pour prendre en charge ces scénarios, des *tuples* ont été ajoutées à C#. Les tuples sont des structures de données légères contenant plusieurs champs pour représenter les membres de données. Les champs ne sont pas validés et vous ne pouvez pas définir vos propres méthodes. Les types de tuple C# prennent en charge == et `!=`

Vous pouvez créer un tuple en assignant une valeur à chaque membre et éventuellement, en fournissant des noms sémantiques à chacun des membres du tuple :

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

Le tuple `namedLetters` contient des champs appelés `Alpha` et `Beta`. Ces noms existent uniquement au moment de la compilation et ne sont pas conservés, par exemple lors de l'inspection du tuple à l'aide de la réflexion au moment de l'exécution.

Dans une assignation de tuple, vous pouvez également spécifier les noms des champs dans la partie droite :

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

Dans certains cas, vous pouvez souhaiter décompresser les membres d'un tuple qui ont été retournés à partir d'une méthode. Pour ce faire, déclarez des variables distinctes pour chacune des valeurs dans le tuple. Cette décompression est appelée *déconstruction* du tuple :

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

Vous pouvez également fournir une déconstruction similaire pour tout type dans le .NET. Vous écrivez une méthode `Deconstruct` en tant que membre de la classe. Cette méthode `Deconstruct` fournit un ensemble d'arguments `out` pour chacune des propriétés que vous voulez extraire. Prenons la classe `Point` suivante qui fournit une méthode de déconstructeur qui extrait les coordonnées `X` et `Y` :

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

Vous pouvez extraire les champs individuels en affectant un `Point` à un tuple :

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

Bien souvent, lorsque vous initialisez un tuple, les variables utilisées pour le côté droit de l'assignation sont les mêmes que les noms que vous souhaitez pour les éléments de tuple : les noms des éléments de tuple peuvent être déduits à partir des variables utilisées pour initialiser le tuple :

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

Souvent, lors de la déconstruction d'un tuple ou de l'appel d'une méthode avec des paramètres `out`, vous devez définir une variable dont la valeur ne vous importe pas et que vous ne prévoyez pas d'utiliser. C# ajoute la prise en charge des *éléments ignorés* pour gérer ce scénario. Un élément ignoré est une variable en écriture seule dont le nom est _ (caractère de soulignement) ; vous pouvez assigner toutes les valeurs que vous souhaitez ignorer à la variable unique. Un élément ignoré est semblable à une variable non assignée ; en dehors de l'instruction d'assignation, l'élément ignoré ne peut pas être utilisé dans le code.

Les éléments ignorés sont pris en charge dans les scénarios suivants :

- Lors de la déconstruction de tuples ou de types définis par l'utilisateur.
- Lors d'appels à des méthodes avec des paramètres out.
- Dans une opération de critères spéciaux avec les instructions is et switch.
- Comme un identificateur autonome quand vous voulez explicitement identifier la valeur d'une assignation comme un élément ignoré.

L'exemple suivant définit une méthode `QueryCityDataForYears` qui retourne un tuple à 6 composants qui contient des données pour une ville au cours de deux années différentes. L'appel de méthode dans l'exemple s'intéresse uniquement à deux valeurs de population retournées par la méthode et traite par conséquent les valeurs restantes dans le tuple comme des éléments ignorés lors de la déconstruction du tuple.

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

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York
City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
```

```
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//       Population change, 1960 to 2010: 393,149
```

4.1.2 Critères spéciaux

Les *critères spéciaux* sont un ensemble de fonctionnalités qui permettent de nouvelles façons d'exprimer le contrôle du workflow dans votre code. Vous pouvez tester des variables pour leur type, leurs valeurs ou les valeurs de leurs propriétés. Ces techniques créent un flot de code plus lisible.

Les critères spéciaux prennent en charge les expressions `is` et les expressions `switch`. L'une ou l'autre permettent d'inspecter un objet et ses propriétés afin de déterminer s'il correspond au modèle recherché. Vous utilisez le mot clé `when` pour spécifier des règles supplémentaires pour le modèle.

L' `is` expression de modèle étend l' is opérateur familier pour interroger un objet sur son type et assigner le résultat dans une instruction. Le code suivant vérifie si une variable est un `int`, et si tel est le cas, il l'ajoute à la somme actuelle :

```
if (input is int count)
    sum += count;
```

Le petit exemple précédent illustre les améliorations apportées à l'expression is. Vous pouvez tester par rapport à des types valeur, ainsi que des types référence, et vous pouvez assigner le résultat de réussite à une nouvelle variable du type correct.

L'expression de correspondance switch a une syntaxe familière, basée sur l'instruction switch qui fait déjà partie du langage C#. L'instruction switch mise à jour a plusieurs nouvelles constructions :

- Le type directeur d'une expression switch n'est plus limité aux types intégraux, aux types Enum, string ou à un type nullable correspondant à l'un de ces types. Vous pouvez utiliser n'importe quel type.
- Vous pouvez tester le type de l'expression switch dans chaque étiquette case. Comme avec l'expression is, vous pouvez assigner une nouvelle variable à ce type.
- Vous pouvez ajouter une clause when pour tester encore les conditions sur cette variable.
- L'ordre des étiquettes case est à présent important. La première branche à faire correspondre est exécutée ; les autres sont ignorées.

Le code suivant illustre ces nouvelles fonctionnalités :

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` est un modèle de constante.
- `case IEnumerable<int> childSequence:` est un modèle de déclaration.
- `case int n when n > 0:` est un modèle de déclaration avec une `when` condition supplémentaire.
- `case null:` est le `null` modèle de constante.
- `default:` est le cas par défaut classique.

À compter de C# 7.1, l'expression de modèle de `is` et du modèle de type `switch` peut avoir le type d'un paramètre de type générique, ce qui peut se révéler particulièrement utile pour vérifier des types potentiellement `struct` ou `class` tout en évitant le boxing.

4.1.3 Async main

Une méthode *async main* vous permet d'utiliser `await` dans votre méthode `Main`. Auparavant, vous deviez écrire :

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

Vous pouvez désormais écrire :

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

Si votre programme ne retourne pas de code de sortie, vous pouvez déclarer une méthode `Main` qui retourne une Task :

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

4.1.4 Fonctions locales

De nombreuses conceptions pour les classes incluent des méthodes qui sont appelées à partir d'un seul emplacement. Ces méthodes privées supplémentaires maintiennent chaque méthode réduite et focalisée. Les *fonctions locales* vous permettent de déclarer des méthodes dans le contexte d'une autre méthode. Il est ainsi plus facile pour les lecteurs de la classe de voir que la méthode locale est appelée uniquement à partir du contexte dans lequel elle a été déclarée.

Il existe deux cas d'utilisation courants pour les fonctions locales : les méthodes iterator publiques et les méthodes async publiques. Ces deux types de méthodes génèrent du code qui signale les erreurs plus tard que ce qu'attendent les programmeurs. Dans les méthodes iterator, toute exception est observée uniquement lors de l'appel de code qui énumère la séquence retournée. Dans les méthodes async, toute exception est observée uniquement quand le `Task` retourné est attendu. L'exemple suivant illustre la séparation entre la validation de paramètres et l'implémentation de l'itérateur à l'aide d'une fonction locale :

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start),
message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end),
message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than
{nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

Il est possible d'utiliser la même technique avec les méthodes `async` pour garantir que les exceptions résultant de la validation d'argument sont levées avant le début de la tâche asynchrone :

```
public Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    }
}
```

Cette syntaxe est maintenant prise en charge :

```
field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

L'attribut `SomeThingAboutFieldAttribute` est appliqué au champ de stockage généré par le compilateur pour `SomeProperty`.

4.1.5 Autres membres expression-bodied

C# 6 a introduit les membres des expressions et des propriétés en lecture seule. C# 7.0 développe les membres autorisés pouvant être implémentés comme expressions. Dans C# 7.0, vous pouvez implémenter des *constructeurs*, des *finaliseurs* ainsi que des accesseurs `get` et `set` sur des *propriétés* et des *indexeurs*. Le code suivant présente des exemples de chaque élément :

```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

4.1.6 Expressions throw

En C#, `throw` a toujours été une instruction. Étant donné que `throw` est une instruction, et non pas une expression, certaines constructions C# ne pouvaient pas l'utiliser. Il s'agit notamment des expressions conditionnelles, des expressions de fusion null, ainsi que de certaines expressions lambda. L'ajout de membres expression-bodied ajoute des emplacements supplémentaires où les expressions `throw` seraient utiles. Afin que vous puissiez écrire l'une de ces constructions, C# 7,0 introduit des expressions Throw.

4.1.7 Expressions littérales par défaut

Les expressions littérales par défaut sont une amélioration des expressions de valeur par défaut. Ces expressions initialisent une variable à la valeur par défaut. Vous deviez écrire auparavant :

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

Vous pouvez désormais omettre le type du côté droit de l'initialisation :

```
Func<string, bool> whereClause = default;
```

4.1.8 Améliorations de la syntaxe littérale numérique

La mauvaise interprétation de constantes numériques peut rendre plus difficile la compréhension du code quand il est lu pour la première fois. Les masques de bits ou d'autres valeurs symboliques sont source de méprise. C# 7.0 comprend deux nouvelles fonctionnalités permettant d'écrire des nombres de la manière la plus lisible pour l'utilisation prévue : les *littéraux binaires* et les *séparateurs de chiffres*.

Dans les cas où vous créez des masques de bits chaque fois qu'une représentation binaire d'un nombre rend le code plus lisible, écrivez ce nombre au format binaire :

```
public const int Sixteen =   0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

Le `0b` au début de la constante indique que le nombre est écrit sous la forme d'un nombre binaire. Les nombres binaires peuvent être longs. il est donc souvent plus facile de voir les modèles binaires en introduisant le _ sous forme de séparateur numérique, comme indiqué dans la constante binaire de l'exemple précédent. Le séparateur de chiffres peut apparaître n'importe où dans la constante. Pour les nombres de base 10, il est courant de l'utiliser comme séparateur des milliers. Les littéraux numériques hexadécimaux et binaires peuvent commencer par un _ :

```
public const long BillionsAndBillions = 100_000_000_000;
```

Il est possible d'utiliser le séparateur de chiffres également avec les types `decimal`, `float` et `double` :

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio =
1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

4.1.9 Variables out

La syntaxe existante qui prend en charge les `out` paramètres a été améliorée en C# 7. Vous pouvez désormais déclarer des variables `out` dans la liste d'arguments d'un appel de méthode, au lieu d'écrire une instruction de déclaration distincte :

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

Vous pouvez spécifier le type de la `out` variable pour plus de clarté, comme indiqué dans l'exemple précédent. Toutefois, le langage prend en charge l'utilisation d'une variable locale implicitement typée :

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- Le code est plus facile à lire.
    - Vous déclarez la variable out là où vous l'utilisez, et non sur une ligne de code précédente.
- Il n'est pas nécessaire d'assigner une valeur initiale.
    - En déclarant la variable `out` à l'endroit où elle est utilisée dans un appel de méthode, vous ne pouvez pas l'utiliser accidentellement avant qu'elle soit assignée.

La syntaxe ajoutée dans C# 7.0 pour autoriser les déclarations variables `out` a été étendue pour inclure des initialiseurs de champ, des initialiseurs de propriété, des initialiseurs de constructeur et des clauses de requête. Il permet d'écrire un code tel que l'exemple suivant :

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

4.1.10 private protected

Enfin, un nouveau modificateur d'accès composé, `private protected`, indique qu'un membre est accessible à la classe globale ou aux classes dérivées déclarées dans le même assembly. Alors que `protected internal` autorise l'accès par des classes dérivées ou qui se trouvent dans le même assembly, `private protected` limite l'accès aux types dérivés déclarés dans le même assembly.

4.1.11 variables locales et retours ref

Cette fonctionnalité active les algorithmes qui utilisent et retournent des références à des variables définies ailleurs. C'est le cas, par exemple, lors de la recherche d'un emplacement unique comportant certaines caractéristiques dans une matrice de grande taille. La méthode suivante retourne une référence à ce stockage dans la matrice :

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Vous pouvez déclarer la valeur de retour en tant que `ref` et modifier cette valeur dans la matrice, comme indiqué dans le code suivant :

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

Le langage C# a plusieurs règles qui vous protègent contre une mauvaise utilisation des variables locales et des retours ref :

- Vous devez ajouter le mot clé ref à la signature de méthode et à toutes les instructions return dans une méthode.
    - Cela permet de clarifier le retour par référence tout au long de la méthode.
- Un ref return peut être assigné à une variable de la valeur ou à une variable ref.
    - L'appelant contrôle si la valeur de retour est copiée ou non. L'omission du modificateur ref lors de l'assignation de la valeur de retour indique que l'appelant souhaite une copie de la valeur, pas une référence au stockage.
- Vous ne pouvez pas affecter une valeur de retour de méthode standard à une variable locale ref.
    - Cela rejette les instructions telles que ref int i = sequence.Count();.
- Vous ne pouvez pas retourner un ref à une variable dont la durée de vie ne s'étend pas au-delà de l'exécution de la méthode.
    - Cela signifie que vous ne pouvez pas retourner une référence à une variable locale ni une variable avec une étendue similaire.
- Les variables locales et les retours ref ne peuvent pas être utilisés avec les méthodes Async.
    - Le compilateur ne peut pas savoir si la variable référencée a été définie à sa valeur finale quand la méthode Async est retournée.

L'ajout de variables locales ref et de retours ref permet d'utiliser des algorithmes qui sont plus efficaces en évitant la copie de valeurs, ou d'effectuer plusieurs fois des opérations de déréférencement.

L'ajout de `ref` à la valeur de retour est une modification compatible avec la source. Le code existant est compilé, mais la valeur de retour référencée est copiée lorsqu'elle est assignée. Les appelants doivent mettre à jour le stockage pour la valeur de retour sur une variable locale `ref` afin de stocker la valeur de retour en tant que référence.

Désormais, les variables locales `ref` peuvent être réassignées pour faire référence à d'autres instances, après avoir été initialisées. Le code suivant effectue maintenant une compilation :

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to
different storage.
```

4.1.12 Expressions ref conditionnelles

Enfin, l'expression conditionnelle peut produire comme résultat une référence plutôt qu'une valeur. Prenons par exemple la ligne suivante, qui récupère une référence au premier élément dans un des deux tableaux :

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

La variable `r` est une référence à la première valeur de `arr` ou `otherArr`.

4.1.13 modification de paramètre du in

Le `in` mot clé complète les mots clés REF et out existants pour passer des arguments par référence. Le mot clé `in` spécifie que l'argument doit être passé par référence, mais la méthode appelée ne modifie pas la valeur.

Vous pouvez déclarer des surcharges qui passent par valeur ou par référence ReadOnly, comme indiqué dans le code suivant :

```
static void M(S arg);
static void M(in S arg);
```

La valeur par valeur (tout d'abord dans l'exemple précédent) est meilleure que la version de référence par ReadOnly. Pour appeler la version avec l'argument de référence en lecture seule, vous devez inclure le modificateur `in` lors de l'appel de la méthode.

4.1.14 D'autres types prennent en charge l'instruction fixed

L'instruction `fixed` prenait en charge un ensemble limité de types. À partir de C# 7.3, tout type contenant une méthode `GetPinnableReference()` qui retourne `ref T` ou `ref readonly T` peut être `fixed`. L'ajout de cette fonctionnalité signifie que `fixed` peut être utilisé avec System.Span<T> et des types connexes.

4.1.15 L'indexation des champs `fixed` ne nécessite aucun épinglage

Prenons le struct suivant :

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

Dans les versions antérieures de C#, vous deviez épingler une variable pour accéder à un des entiers faisant partie de `myFixedField`. À présent, le code suivant se compile sans épingler la variable `p` à l'intérieur d'une instruction `fixed` distincte :

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

La variable `p` accède à un élément dans `myFixedField`. Vous n'avez pas besoin de déclarer une variable `int*` distincte. Vous avez toujours besoin d'un `unsafe` contexte. Dans les versions antérieures de C#, vous devez déclarer un second pointeur fixe :

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

4.1.16 Les tableaux `stackalloc` prennent en charge les initialiseurs

Vous avez été en mesure de spécifier les valeurs des éléments dans un tableau lors de son initialisation :

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

Maintenant, la même syntaxe peut être appliquée aux tableaux déclarés avec `stackalloc` :

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

4.1.17 Contraintes génériques améliorées

Vous pouvez maintenant spécifier le type System.Enum ou System.Delegate en tant que contraintes de classe de base pour un paramètre de type.

Vous pouvez également utiliser la nouvelle `unmanaged` contrainte pour spécifier qu'un paramètre de type doit être un type non managé qui n'accepte pas les valeurs NULL.

Pour plus d'informations, consultez les articles sur les contraintes et contraintes where génériques sur les paramètres de type.

L'ajout de ces contraintes aux types existants est une modification managé. Les types génériques fermés peuvent ne plus respecter ces nouvelles contraintes.

4.1.18 Types de retour async généralisés

Le retour d'un objet `Task` à partir de méthodes async peut introduire des goulots d'étranglement au niveau des performances dans certains chemins. `Task` est un type référence. Si vous l'utilisez, vous allouez donc un objet. Dans les cas où une méthode déclarée avec le modificateur `async` retourne un résultat mis en cache, ou si elle s'exécute de manière synchrone, le coût en termes de temps induit par les allocations supplémentaires peut s'avérer significatif dans les sections de code critiques pour les performances. Cela peut devenir coûteux si ces allocations se produisent dans des boucles serrées.

La nouvelle fonctionnalité du langage signifie que les types de retour des méthodes async ne se limitent pas à `Task`, `Task<T>` et `void`. Le type retourné doit toujours correspondre au modèle async, ce qui signifie qu'une méthode `GetAwaiter` doit être accessible. En guise d'exemple concret, le `ValueTask` type a été ajouté à .net pour utiliser cette nouvelle fonctionnalité de langage :

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
    return 5;
}
```

4.2 C# 8.0

C# 8,0 est pris en charge sur **.net Core 3. x** et **.NET standard 2,1.**

4.2.1 Membres ReadOnly

Vous pouvez appliquer le `readonly` modificateur aux membres d'un struct. Elle indique que le membre ne modifie pas l'État. C'est plus précis que d'appliquer le modificateur `readonly` à une déclaration `struct`. Examinons le struct mutable suivant :

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

Comme la plupart des structs, la `ToString()` méthode ne modifie pas l'État. Vous pouvez indiquer cela en ajoutant le modificateur `readonly` à la déclaration de `ToString()` :

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

La modification précédente génère un avertissement du compilateur, car `ToString` accède à la `Distance` propriété, qui n'est pas marquée comme `readonly` suit :

```
 warning CS8656: Call to non-readonly member 'Point.Distance.get' from a
'readonly' member results in an implicit copy of 'this'
```

Le compilateur vous avertit lorsqu'il a besoin de créer une copie défensive. La `Distance` propriété ne change pas d'État. vous pouvez donc résoudre cet avertissement en ajoutant le `readonly` modificateur à la déclaration :

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Notez que le `readonly` modificateur est nécessaire sur une propriété en lecture seule. Le compilateur ne suppose pas que les `get` accesseurs ne modifient pas l'état ; vous devez déclarer `readonly` explicitement. Les propriétés implémentées automatiquement sont une exception. le compilateur traite tous les accesseurs get implémentés automatiquement comme. `readonly` il n'est donc pas nécessaire d'ajouter le `readonly` modificateur aux `X` `Y` Propriétés et.

Le compilateur applique la règle qui `readonly` ne modifie pas l'état des membres. La méthode suivante n'est pas compilée, sauf si vous supprimez le `readonly` modificateur :

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

4.2.2 Expressions switch

Souvent, une switch instruction produit une valeur dans chacun de ses `case` blocs. **Les expressions switch** permettent d'utiliser une syntaxe d'expression plus concise, comprenant moins de mots clés `case` et `break` répétitifs et moins d'accolades. Prenons par exemple l'enum suivant, qui liste les couleurs de l'arc-en-ciel :

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

Si votre application a défini un type `RGBColor` construit à partir des composants `R`, `G` et `B`, vous pouvez convertir une valeur `Rainbow` en valeurs RVB avec la méthode suivante, qui contient une expression switch :

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid
enum value", paramName: nameof(colorBand)),
    };
```

Plusieurs améliorations ont ici été apportées à la syntaxe :

- La variable se situe avant le mot clé switch. Ce nouvel ordre permet de distinguer visuellement l'expression switch de l'instruction switch.
- Les éléments case et : sont remplacés par =>, plus concis et plus intuitif.
- Le cas default est remplacé par un discard _.
- Les corps sont des expressions et non des instructions.

Comparez cela avec le code équivalent utilisant l'instruction switch classique :

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value",
paramName: nameof(colorBand));
    };
}
```

4.2.3 Modèles de propriétés

Le **modèle de propriété** permet de faire correspondre les propriétés de l'objet examiné. Prenons un site d'e-commerce qui doit calculer les taxes sur les ventes en fonction de l'adresse de l'acheteur. Ce calcul n'est pas une responsabilité fondamentale d'une `Address` classe. Il changera au fil du temps, probablement plus souvent que n'évoluera le format de l'adresse. Le montant des taxes sur les ventes varie selon la propriété `State` de l'adresse. La méthode suivante utilise le modèle de propriété pour calculer les taxes sur les ventes à partir de l'adresse et du prix :

```
public static decimal ComputeSalesTax(Address location, decimal salePrice)
=>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

4.2.4 Modèles de tuples

Certains algorithmes dépendent de plusieurs entrées. Les **modèles de tuples** permettent de basculer des unes aux autres en fonction de plusieurs valeurs exprimées sous forme de tuple. Le code suivant montre une expression switch pour le jeu *Pierre-papier-ciseaux* :

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors
wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

4.2.5 Modèles positionnels

Certains types comportent une méthode `Deconstruct` qui décompose ses propriétés en variables discrètes. Lorsqu'une méthode `Deconstruct` est accessible, il est possible d'utiliser des **modèles positionnels** pour inspecter les propriétés de l'objet et de se servir de ces dernières pour un modèle. Considérez la classe `Point` suivante, dont la méthode `Deconstruct` sert à créer des variables discrètes pour `X` et `Y` :

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
```

```
}
```

De plus, envisagez l'énumération suivante qui représente diverses positions d'un quadrant :

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

La méthode suivante utilise le **modèle positionnel** pour extraire les valeurs de `x` et de `y`. Ensuite, elle utilise une clause `when` pour déterminer le `Quadrant` du point :

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

Le modèle discard dans le switch précédent indique une correspondance lorsque soit `x`, soit `y` est égal à 0, mais pas les deux. Une expression switch doit produire une valeur ou, si aucun des cas ne correspond, lever une exception. Le compilateur génère un avertissement pour vous si vous ne traitez pas tous les cas possibles dans votre expression de commutateur.

4.2.6 Déclarations using

Une **déclaration using** est une déclaration de variable précédée par le mot clé `using`. Elle indique au compilateur que la variable déclarée doit être supprimée à la fin de la portée englobante. Prenons par exemple le code suivant, qui écrit un fichier texte :

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
```

```
    // file is disposed here
}
```

Dans l'exemple précédent, le fichier est supprimé une fois l'accolade fermante de la méthode atteinte. C'est la fin de la portée dans laquelle `file` est déclaré. Le code précédent équivaut au suivant, qui utilise l'instruction using classique :

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

Dans l'exemple précédent, le fichier est supprimé une fois l'accolade fermante associée à l'instruction `using` atteinte.

4.2.7 Fonctions locales statiques

Vous pouvez maintenant ajouter le `static` modificateur aux fonctions locales pour vous assurer que la fonction locale ne capture pas (référence) les variables de la portée englobante. Cela génère `CS8421`, « A static local function can't contain a reference to <variable> ».

Prenons le code suivant. La fonction locale `LocalFunction` accède à la variable `y`, déclarée dans la portée englobante (la méthode `M`). Par conséquent, `LocalFunction` ne peut pas être déclarée avec le modificateur `static` :

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

Le code suivant contient une fonction locale statique. Elle peut être statique, car elle n'accède à aucune variable dans la portée englobante :

```
int M()
```

```
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

4.2.8 Flux asynchrones

À partir de C# 8.0, il est possible de créer et de consommer des flux de façon asynchrone. Une méthode qui retourne un flux asynchrone comporte trois propriétés :

a) Elle est déclarée avec le modificateur `async`.
b) Elle retourne un IAsyncEnumerable<T>.
c) La méthode contient des instructions `yield return` pour retourner des éléments consécutifs dans le flux asynchrone.

Pour pouvoir consommer un flux asynchrone, il faut ajouter le mot clé `await` avant le mot clé `foreach` au moment d'énumérer les éléments du flux. Le mot clé `await` exige que la méthode qui énumère le flux asynchrone soit déclarée avec le modificateur `async` et retourne un type autorisé pour une méthode `async`, soit en général Task ou Task<TResult>. Il peut également s'agir de ValueTask ou ValueTask<TResult>. Une méthode peut consommer et produire un flux asynchrone ; elle retournerait alors un IAsyncEnumerable<T>. Le code suivant génère une séquence de 0 à 19, en attendant 100 ms entre chaque nombre :

```
public static async System.Collections.Generic.IAsyncEnumerable<int>
GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

Pour énumérer la séquence, on utilise l'instruction `await foreach` :

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

4.2.9 Index et plages

Les index et les plages fournissent une syntaxe concise pour accéder à des éléments ou des plages uniques dans une séquence.

Cette prise en charge de langage s'appuie sur deux nouveaux types et deux nouveaux opérateurs :

- System.Index représente un index au sein d'une séquence.
- Index de l'opérateur End `^` , qui spécifie qu'un index est relatif à la fin de la séquence.
- System.Range représente une sous-plage d'une séquence.
- Opérateur de plage `..` , qui spécifie le début et la fin d'une plage comme opérandes.

Commençons par les règles concernant les index. Prenons pour exemple un tableau `sequence`. L'index `0` est identique à l'index `sequence[0]`. L'index `^0` est identique à l'index `sequence[sequence.Length]`. Notez que `sequence[^0]` lève une exception, tout comme `sequence[sequence.Length]`. Pour n'importe quel nombre `n`, l'index `^n` est identique à l'index `sequence.Length - n`.

Une plage spécifie son *début* et sa *fin*. Le début de la plage est compris, mais la fin de la plage est exclusive, ce qui signifie que le *début* est inclus dans la plage, mais que la *fin* n'est pas incluse dans la plage. La plage `[0..^0]` représente la plage dans son intégralité, tout comme `[0..sequence.Length]` représente la plage entière.

Prenons quelques exemples. Examinez le tableau suivant, annoté avec son index à partir du début et de la fin :

```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Vous pouvez récupérer le dernier mot avec l'index `^1` :

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

Le code suivant crée une sous-plage qui comporte les mots « quick », « brown » et « fox » et va de `words[1]` à `words[3]`. L'élément `words[4]` n'est pas dans la plage.

```
var quickBrownFox = words[1..4];
```

Le code suivant crée une sous-plage qui comporte « lazy » et « dog » et comprend `words[^2]` et `words[^1]`. L'index de fin `words[^0]` n'est pas inclus :

```
var lazyDog = words[^2..^0];
```

Les exemples suivants créent des plages ouvertes au début, à la fin ou les deux :

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

Vous pouvez également déclarer des plages comme variables :

```
Range phrase = 1..4;
```

La plage peut ensuite être utilisée à l'intérieur des caractères `[` et `]` :

```
var text = words[phrase];
```

Non seulement les tableaux prennent en charge les index et les plages. Vous pouvez également utiliser des index et des plages avec String, Span<T> ou ReadOnlySpan<T>. Pour plus d'informations, consultez prise en charge des types d'index et de plages.

4.2.10 Assignation de fusion Null

C# 8,0 introduit l'opérateur d'assignation de fusion Null `??=` . Vous pouvez utiliser l' `??=` opérateur pour assigner la valeur de son opérande droit à son opérande gauche uniquement si l'opérande de gauche prend la valeur `null` .

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

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers));  // output: 17 17
Console.WriteLine(i);  // output: 17
```

4.2.11 Types construits non managés

En C# 7,3 et versions antérieures, un type construit (un type qui comprend au moins un argument de type) ne peut pas être un type non managé. À compter de C# 8,0, un type valeur construit est non managé s'il contient uniquement des champs de types non managés.

Par exemple, étant donné la définition suivante du `Coords<T>` type générique :

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

le `Coords<int>` type est un type non managé dans C# 8,0 et versions ultérieures. Comme pour tout type non managé, vous pouvez créer un pointeur vers une variable de ce type ou allouer un bloc de mémoire sur la pile pour les instances de ce type :

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

4.2.12 Stackalloc dans les expressions imbriquées

À compter de C# 8,0, si le résultat d'une expression stackalloc[2] est du System.Span<T> System.ReadOnlySpan<T> type ou, vous pouvez utiliser l' `stackalloc` expression dans d'autres expressions :

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

4.2.13 Amélioration des chaînes textuelles interpolées

L'ordre des `$ @` jetons et dans les chaînes textuelles interpolées peut être any : `$@"..."` et `@$"..."` sont des chaînes textuelles interpolées valides. Dans les versions antérieures de C#, le `$` jeton doit apparaître avant le `@` jeton.

4.3 C# 9.0

C# 9,0 est pris en charge sur **.net 5**.

4.3.1 Types d'enregistrements

C# 9,0 introduit les ***types d'enregistrements***. Vous utilisez le `record` mot clé pour définir un type de référence qui fournit des fonctionnalités intégrées pour l'encapsulation des données. Vous pouvez créer des types d'enregistrements avec des propriétés immuables à l'aide de paramètres positionnels ou d'une syntaxe de propriété standard :

```
public record Person(string FirstName, string LastName);


public record Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
};
```

Vous pouvez également créer des types d'enregistrements avec des propriétés et des champs mutables :

```
public record Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
};
```

4.3.2 Syntaxe de position pour la définition de propriété

Vous pouvez utiliser des paramètres positionnels pour déclarer les propriétés d'un enregistrement et initialiser les valeurs de propriété lors de la création d'une instance :

[2] Une `stackalloc` expression alloue un bloc de mémoire sur la pile. Un bloc de mémoire alloué dans la pile pendant l'exécution de la méthode est automatiquement supprimé lorsque cette méthode retourne un résultat. Vous ne pouvez pas libérer explicitement la mémoire allouée avec `stackalloc` . Un bloc de mémoire alloué à la pile n'est pas soumis à garbage collection et ne doit pas être épinglé à une fixed instruction.

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

Lorsque vous utilisez la syntaxe de position pour la définition de propriété, le compilateur crée :

- Propriété publique init-only implémentée automatiquement pour chaque paramètre positionnel fourni dans la déclaration d'enregistrement. Une propriété init-only ne peut être définie que dans le constructeur ou à l'aide d'un initialiseur de propriété.
- Constructeur principal dont les paramètres correspondent aux paramètres positionnels de la déclaration d'enregistrement.
- Une `Deconstruct` méthode avec un `out` paramètre pour chaque paramètre positionnel fourni dans la déclaration d'enregistrement.

4.3.3 Immuabilité

Un type d'enregistrement n'est pas nécessairement immuable. Vous pouvez déclarer des propriétés avec des `set` accesseurs et des champs qui ne le sont pas `readonly` . Toutefois, bien que les enregistrements puissent être mutables, ils facilitent la création de modèles de données immuables. Les propriétés que vous créez à l'aide de la syntaxe positionnel sont immuables.

L'immuabilité peut être utile lorsque vous souhaitez qu'un type centré sur les données soit thread-safe ou un code de hachage pour rester le même dans une table de hachage. Elle peut empêcher les bogues qui se produisent lorsque vous transmettez un argument par référence à une méthode, et que la méthode modifie de manière inattendue la valeur de l'argument.

Les fonctionnalités propres aux types d'enregistrements sont implémentées par les méthodes synthétisées par le compilateur, et aucune de ces méthodes ne compromet l'immuabilité en modifiant l'état de l'objet.

4.3.4 Égalité des valeurs

L'égalité des valeurs signifie que deux variables d'un type d'enregistrement sont égales si les types correspondent et que toutes les valeurs de propriété et de champ correspondent. Pour les autres types de référence, l'égalité correspond à l'identité. Autrement dit, deux variables d'un type référence sont égales si elles font référence au même objet.

L'exemple suivant illustre l'égalité des valeurs des types d'enregistrements :

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
```

```
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

Dans `class` les types, vous pouvez substituer manuellement les méthodes et les opérateurs d'égalité pour atteindre l'égalité des valeurs, mais le développement et le test de ce code seraient fastidieux et sujets aux erreurs. Le fait de disposer de cette fonctionnalité permet d'éviter les bogues susceptibles d'oublier de mettre à jour le code de remplacement personnalisé lors de l'ajout ou de la modification des propriétés ou des champs.

4.3.5 Mutation non destructrice

Si vous devez muter des propriétés immuables d'une instance d'enregistrement, vous pouvez utiliser une `with` expression pour obtenir une *mutation non destructrice*. Une `with` expression crée une nouvelle instance d'enregistrement qui est une copie d'une instance d'enregistrement existante, avec les propriétés et les champs spécifiés modifiés. Utilisez la syntaxe de l' initialiseur d'objet pour spécifier les valeurs à modifier, comme indiqué dans l'exemple suivant :

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1]
};
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers
= System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

4.3.6 Mise en forme intégrée pour l'affichage

Les types d'enregistrements ont une méthode générée par ToString le compilateur qui affiche les noms et les valeurs des propriétés et des champs publics. La `ToString` méthode retourne une chaîne au format suivant :

<record type name> { <property name> = <value>, <property name> = <value>, ...}

Pour les types référence, le nom de type de l'objet auquel la propriété fait référence s'affiche à la place de la valeur de la propriété. Dans l'exemple suivant, le tableau est un type référence, donc `System.String[]` s'affiche à la place des valeurs d'élément de tableau réelles :

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames =
System.String[] }
```

4.3.7 Héritage

Un enregistrement peut hériter d'un autre enregistrement. Toutefois, un enregistrement ne peut pas hériter d'une classe, et une classe ne peut pas hériter d'un enregistrement.

L'exemple suivant illustre l'héritage avec la syntaxe de propriété de position :

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Pour que deux variables d'enregistrement soient égales, le type au moment de l'exécution doit être égal. Les types des variables conteneur peuvent être différents. Cela est illustré dans l'exemple de code suivant :

```
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

Dans l'exemple, toutes les instances ont les mêmes propriétés et les mêmes valeurs de propriété. Mais `student == teacher` retourne `False` même si les deux sont des `Person` variables de type. Et `student == student2` retourne la valeur `True` même si l'une est une `Person` variable et l'autre une `Student` variable.

Toutes les propriétés publiques et les champs des types dérivés et de base sont inclus dans la `ToString` sortie, comme illustré dans l'exemple suivant :

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
```

```
}
```

4.3.8 Setter init uniquement

Les ***accesseurs set init only*** fournissent une syntaxe cohérente pour initialiser les membres d'un objet. Les initialiseurs de propriété permettent de savoir clairement quelle valeur définit la propriété. L'inconvénient est que ces propriétés doivent pouvoir être définies. À compter de C# 9,0, vous pouvez créer des `init` accesseurs au lieu d' `set` accesseurs pour les propriétés et les indexeurs. Les appelants peuvent utiliser la syntaxe de l'initialiseur de propriété pour définir ces valeurs dans les expressions de création, mais ces propriétés sont en lecture seule une fois que la construction est terminée. Les Setters init uniquement fournissent une fenêtre pour modifier l'État. Cette fenêtre se ferme à la fin de la phase de construction. La phase de construction se termine effectivement après toute initialisation, y compris les initialiseurs de propriété et les expressions with-.

Vous pouvez déclarer `init` uniquement des accesseurs set dans n'importe quel type que vous écrivez. Par exemple, le struct suivant définit une structure d'observation météorologique :

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars}
pressure";
}
```

Les appelants peuvent utiliser la syntaxe de l'initialiseur de propriété pour définir les valeurs, tout en préservant l'immuabilité :

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

Une tentative de modification d'une observation après l'initialisation provoque une erreur du compilateur :

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Les Setters init uniquement peuvent être utiles pour définir des propriétés de classe de base à partir de classes dérivées. Ils peuvent également définir des propriétés dérivées par le biais d'assistances dans une classe de base. Les enregistrements positionnels déclarent des propriétés à l'aide des accesseurs set init only. Ces accesseurs set sont utilisés dans les expressions with. Vous pouvez déclarer des accesseurs set init uniquement pour tout `class` , `struct` ou `record` vous définissez.

4.3.9 Instructions de niveau supérieur

Les instructions de niveau supérieur suppriment la cérémonie inutile de nombreuses applications. Prenons le « Hello World ! » canonique. Table.

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Il n'y a qu'une seule ligne de code qui fait quoi que ce soit. Avec les instructions de niveau supérieur, vous pouvez remplacer tout ce qui est utilisé par la `using` directive et la ligne unique qui effectue le travail :

```
using System;

Console.WriteLine("Hello World!");
```

Si vous souhaitez un programme à une seule ligne, vous pouvez supprimer la `using` directive et utiliser le nom de type complet :

```
System.Console.WriteLine("Hello World!");
```

4.3.10 Améliorations des critères spéciaux

C# 9 comprend de nouvelles améliorations de la mise en correspondance des modèles :

- Les ***modèles de type*** correspondent à une variable est un type
- Les ***modèles entre parenthèses*** appliquent ou mettent en évidence la précédence des combinaisons de modèles
- ***Les `and` modèles conjonctives*** nécessitent que les deux modèles correspondent
- ***Les `or` modèles disjonctive*** nécessitent un modèle de correspondance
- Les ***`not` modèles de négation*** requièrent qu'un modèle ne corresponde pas
- Les ***modèles relationnels*** requièrent que l'entrée soit inférieure à, supérieure à, inférieure ou égale à une constante donnée, ou supérieure ou égale à celle-ci.

Ces modèles enrichissent la syntaxe des modèles. Prenons les exemples suivants :

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Avec des parenthèses facultatives pour qu'elle soit claire et qu'elle `and` a une priorité plus élevée que `or` :

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

L’une des utilisations les plus courantes est une nouvelle syntaxe pour un contrôle de valeur NULL :

```
if (e is not null)
{
    // ...
}
```

Chapitre 2 : Environnements .NET

Je vais vous présenter les quatre environnements :

- .NET Framework
- .NET Core
- .NET Standard
- .NET Microsoft

1) .NET Framework

C'est un cadre logiciel qui a été créé par Microsoft en 2002, en même temps que C#, et aussi par la même personne.

Le .NET Framework = CLR + ASP.NET + ADO.NET + Entity Framework + LINQ + WinForms + WCF + WPF+…

Permettant de développer des programmes (applications) fonctionnant dans un environnement Microsoft (sous Windows=.

Plusieurs langages possibles : C#, C++, F#, J#, VB.NET, …

Par exemple :

Paint.net : l'éditeur d'images est réalisé avec .NET Framework.

2) .NET Core

Le .NET Core est un cadre logiciel open-source créé en 2016 pour Windows, macOS et Linux.

Le .NET Core est écrit en C# et C++.

Permettant de développer seulement des

- Applications consoles
- Applications Web
- Microservices

Plusieurs langages supportés : C#, F#, VB.NET (dans la prochaine version), (aucun support pour C++).

Une remarque importante que .NET Core n'est pas une nouvelle version de .NET Framework, il s'agit d'une nouvelle plateforme écrit à partir de zéro.

3) Microsoft .NET

C'est des produits et technologies informatiques de Microsoft.

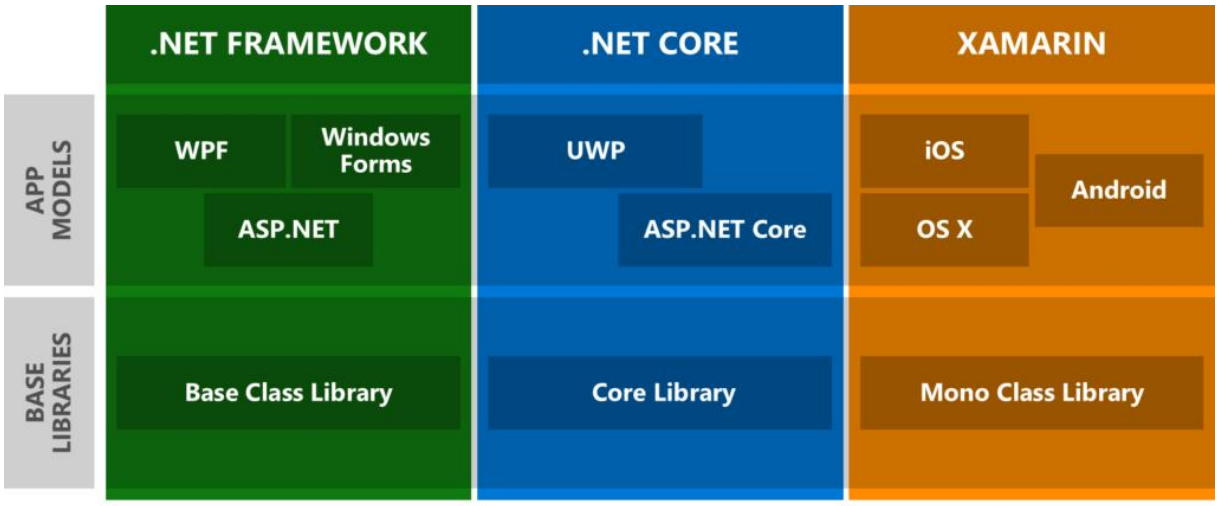
Microsoft .NET = WINDOWS + .NET Framework + Visual Studio + ….

Je note aussi que Visual STUDIO est un IDE (Intergrated Development Environment) qui nous permet d'écrire des programmes avec le . NET Framework.

4) L'environnement .NET permet de créer des applications mobiles :

La Réponse est Oui, en utilisant le framework Xamarin et l'IDE Mono.

5) Résumé final :



Le Framework .NET en plus de prendre en charge plusieurs langages, permet de mettre en place de nombreux types d'applications (WPF, WinForms, ASP.NET WebForms, Azure, etc.).

Une BCL (Base Class Library) pour chaque plateforme.

Est-il possible de développer une bibliothèque de classes dans .NET Framework et l'utiliser dans .NET Core ?

La solution est que dans le .NET Standard, une spécification qui peut être utilisée dans toutes les plateformes .NET : .NET Framework, .NET Core ou Xamarin.

Elle est utilisée uniquement pour le développement de bibliothèques de classes.

Multiplateforme

Il va de soi que si vous développez une application qui devra s'exécuter sur Windows, Linux ou encore OS X, vous devez utiliser .NET Core qui est multiplateforme.

De plus, vous pouvez directement développer votre application en utilisant le système d'exploitation sur lequel cette dernière sera exécutée. Le SDK .NET Core offre de nombreux outils en ligne de commande pouvant être utilisés pour créer, compiler, générer et publier une application .NET Core sous n'importe quel OS.

Pour l'édition de code, Visual Studio Code peut être utilisé. Bien qu'assez éloigné de Visual Studio en termes de fonctionnalités, il a le mérite d'être open source, multiplateforme, léger et offre les fonctionnalités recherchées dans un EDI comme la coloration syntaxique, l'IntelliSence ou encore le débogage. De plus, grâce au projet OmniSharp, les compilateurs pour C# et F#, ainsi que les outils de développement pour .NET Core peuvent s'intégrer avec d'autres éditeurs comme Vim, Atom ou encore Sublime Text.

Performance

Une application ASP.NET Core, par exemple, est dix fois plus rapide que la même application ASP.NET.

Dans un récent billet de blog, Microsoft met en avant les améliorations significatives des performances de .NET Core par rapport au Framework .NET, en ce qui concerne notamment les collections, la compression, la cryptographie, les fonctions mathématiques, la sérialisation, et bien plus.

Cloud et microservices

La modularité de .NET Core permet de publier les applications uniquement avec les bibliothèques qui sont utilisées par l'application. Ce qui réduit de façon drastique la taille d'une application .NET Core. Cela fait donc de la plateforme un client de choix pour les plateformes de cloud et les microservices. Les ressources cloud étant payées à l'utilisation (espace de stockage, CPU, mémoire, etc.), la taille de l'application peut avoir un incident non négligeable sur votre facture.

En microservice, les développeurs pourront mettre en place des petites applications qui s'exécutent side-by side en offrant de meilleures performances qu'avec .NET Framework.

Conteneurs

Docker jouit de nos jours d'une grosse popularité. La plateforme est de plus en plus utilisée par les développeurs pour packager et exécuter leurs applications. Si vous envisagez d'utiliser un conteneur pour exécuter votre application, vous devez choisir .NET Core pour développer cette dernière. Certes il est possible de déployer des applications .NET Framework dans des conteneurs Docker sur Windows, mais la taille de l'image est beaucoup plus importante qu'avec .NET Core et votre image ne pourra être exécutée que sur un hôte Windows.

Une application ASP.NET Core par exemple, est self-hosted, donc ne nécessite d'aucun autre programme supplémentaire pour être exécutée dans un conteneur. De plus, vous pouvez utiliser le serveur Web de votre choix en complément (Apache, NGinx, etc.).

1. Les versions .NET Framework

Je vais vous expliquer les differences entre les versions :

Dans un premier temps, la liste des versions:

.NET Framework 4.8 .NET Framework 4.7.2 .NET Framework 4.7.1 .NET Framework 4.7

.NET Framework 4.6.2 .NET Framework 4.6.1 .NET Framework 4.6 .NET Framework 4.5.2

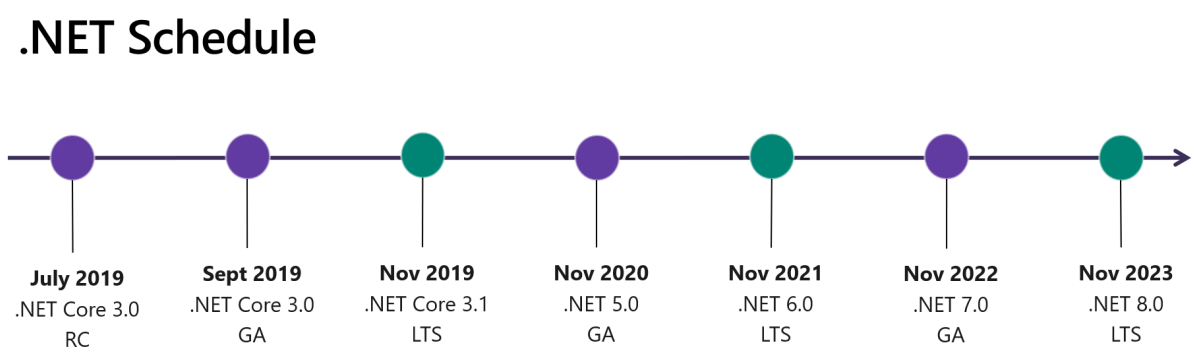
.NET Framework 4.5.1 .NET Framework 4.5 .NET Framework 4 .NET Framework 3.5

.NET Framework 3.0 .NET Framework 2.0 .NET Framework 1.1 .NET Framework 1.0

Je vous donne d'autres differences :

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



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

Descriptions des versions:

- ✓ .NET Framework 4.8

Version CLR	4
Versions de Windows	✔10 mai 2019 mise à jour □ 10 octobre 2018 Update (version 1809) □ 10 avril 2018 mise à jour (version 1803) Mise à jour des créateurs de □ 10 automne (version 1709) □ 10 Creators Update (version 1703) □ 10 mise à jour anniversaire (version 1607) □ 8,1 □ 7
Versions de Windows Server	□ Windows Server 2019 □ Windows Server, version 1809 □ Windows Server, version 1803 □ 2016 □ 2012 R2 □ 2012 □ 2008 R2 SP1

✓ .NET Framework 4.7.2

Version CLR	4
Versions de Windows	Mise à jour des créateurs de ✔10 automne (version 1709) □ 10 Creators Update (version 1703) □ 10 mise à jour anniversaire (version 1607) □ 8,1 □ 7
Versions de Windows Server	□ Windows Server, version 1803 ✔□ Windows Server, version 1709 □ 2016 □ 2012 R2 □ 2012 □ 2008 R2 SP1

✓ .NET Framework 4.7

Version CLR	4
Versions de Windows	✔ 10 Creators Update (version 1703) □ 10 mise à jour anniversaire (version 1607) □ 8,1 □ 7
Versions de Windows Server	□ 2016 □ 2012 R2 □ 2012 □ 2008 R2 SP1

✓ .NET Framework 4.6.2

Version CLR	4
Versions de Windows	✔□ 10 mise à jour anniversaire (version 1607) □ 10 mise à jour de novembre (version 1511) □ 10 □ 8,1 □ 7
Versions de Windows Server	✔□ 2016 □ 2012 R2 □ 2012 □ 2008 R2 SP1

✓ .NET Framework 4.6.1

Version CLR	4
Versions de Windows	✔10 mise à jour de novembre (version 1511) □ 10 □ 8,1 □ 8 □ 7
Versions de Windows Server	□ 2012 R2 □ 2012 □ 2008 R2 SP1

✓ .NET Framework 4.6

Version CLR	4
Versions de Windows	✔□ 10 □ 8,1 □ 8 □ 7 □ Vista
Versions de Windows Server	□ 2012 R2 □ 2012 □ 2008 R2 SP1 □ 2008 SP2

✓ .NET Framework 4.5.2

Version CLR	4
Versions de Windows	□ 8,1 □ 8 □ 7 □ Vista
Versions de Windows Server	□ 2012 R2 □ 2012 □ 2008 R2 SP1 □ 2008 SP2

✓ .NET Framework 4.5.1

Version CLR	4

	✔ 8,1
Inclus dans la version Visual Studio	2013
Versions de Windows	□ 8 □ 7 □ Vista ✔ 2012 R2
Versions de Windows Server	□ 2012 □ 2008 R2 SP1 □ 2008 SP2

✓ .NET Framework 4.5

Version CLR	4
Inclus dans la version Visual Studio	2012
Versions de Windows	✔8 □ 7 □ Vista
Versions de Windows Server	✔ 2012 □ 2008 R2 SP1 □ 2008 SP2

✓ .NET Framework 4

Version CLR	4
Inclus dans la version Visual Studio	2010
Versions de Windows	□ 7 □ Vista
Versions de Windows Server	□ 2008 R2 SP1 □ 2008 SP2 □ 2003

✓ .NET Framework 3.5

- LINQ
- Arborescences de l'expression
- Prise en charge améliorée de ASP.NET pour le développement AJAX
- HashSet (collections)
- DateTimeOffset
- Intégration de WCF et WF
- Réseau pair à pair
- Compléments pour l'extensibilité

Version CLR	2.0
Inclus dans la version Visual Studio	2008
Versions de Windows	✔10* ✔8,1*

Versions de Windows Server	✔ 8* ✔ 7 □ Vista □ Windows Server, version 1803* □ Windows Server, version 1709* □ 2016* □ 2012 R2* □ 2012* ✔2008 R2 SP1* □ 2008 SP2 □ 2003

✓ .NET Framework 3.0

- Windows Presentation Foundation
- Windows Communication Foundation
- Windows Workflow Foundation
- Windows CardSpace

Version CLR	2.0
Versions de Windows	✔Vista
Versions de Windows Server	✔2008 R2 SP1 * ✔ 2008 SP2* □ 2003

✓ .NET Framework 2.0

- Génériques
- Modifier & Continuer du débogueur
- Évolutivité et performances améliorées
- déploiement ClickOnce
- Dans ASP.NET 2,0, nouveaux contrôles et prise en charge d'un large éventail de navigateurs
- prise en charge 64 bits

Version CLR	2.0
Inclus dans la version Visual Studio	2005
Versions de Windows	N/A
Versions de Windows Server	✔2008 R2 SP1 ✔ 2008 SP2

✔ 2003

✓ .NET Framework 1.1

- Contrôles mobiles ASP.NET
- Exécution côte à côte
- Prise en charge d'IPv6

✓ .NET Framework 1.1

Version CLR	1.1
Inclus dans la version Visual Studio	2003
Versions de Windows	N/A
Versions de Windows Server	✔ 2003

✓ .NET Framework 1.0

Version CLR	1.0
Inclus dans la version Visual Studio	Visual Studio .NET
Versions de Windows	N/A
Versions de Windows Server	N/A

2. Les versions .NET Core

Version	Status	Visual Studio 2017 SDK ⓘ	Visual Studio 2019 SDK ⓘ	Runtime ⓘ
.NET 6.0	Preview ⓘ	N/A	x64 SDK | x86 SDK (v6.0.100-preview.1)	x64 Runtime | x86 Runtime (v6.0.0-preview.1)
.NET 5.0	Current ⓘ	N/A	x64 SDK | x86 SDK (v5.0.103)	x64 Runtime | x86 Runtime (v5.0.3)
.NET Core 3.1	LTS ⓘ	N/A	x64 SDK | x86 SDK (v3.1.406)	x64 Runtime | x86 Runtime (v3.1.12)
.NET Core 3.0	End of life ⓘ	N/A	x64 SDK | x86 SDK (v3.0.103)	x64 Runtime | x86 Runtime (v3.0.3)
.NET Core 2.2	End of life ⓘ	x64 SDK | x86 SDK (v2.2.110)	x64 SDK | x86 SDK (v2.2.207)	x64 Runtime | x86 Runtime (v2.2.8)
.NET Core 2.1	LTS ⓘ	x64 SDK | x86 SDK (v2.1.521)	x64 SDK | x86 SDK (v2.1.813)	x64 Runtime | x86 Runtime (v2.1.25)
.NET Core 2.0	End of life ⓘ	x64 SDK | x86 SDK (v2.1.202)	x64 SDK | x86 SDK (v2.1.202)	x64 Runtime | x86 Runtime (v2.0.9)
.NET Core 1.1	End of life ⓘ	x64 SDK | x86 SDK (v1.1.14)	x64 SDK | x86 SDK (v1.1.14)	x64 Runtime | x86 Runtime (v1.1.13)
.NET Core 1.0	End of life ⓘ	x64 SDK | x86 SDK (v1.1.14)	x64 SDK | x86 SDK (v1.1.14)	x64 Runtime | x86 Runtime (v1.0.16)

Chapitre 3 : La plateforme .NET

Le .NET Framework est la nouvelle stratégie de Microsoft pour écrire des applications windows.

La plateforme .NET est compose de plusieurs composants, chacun ayant un rôle bien définit:

- le « Common Language Runtime » ;
- le « Common Intermediate Language (CIL) ;
- le « Common Type System (CTS) ;
- la « Base Class Library » (BCL).

Je vais maintenant présenter le schéma de la plateforme .NET :

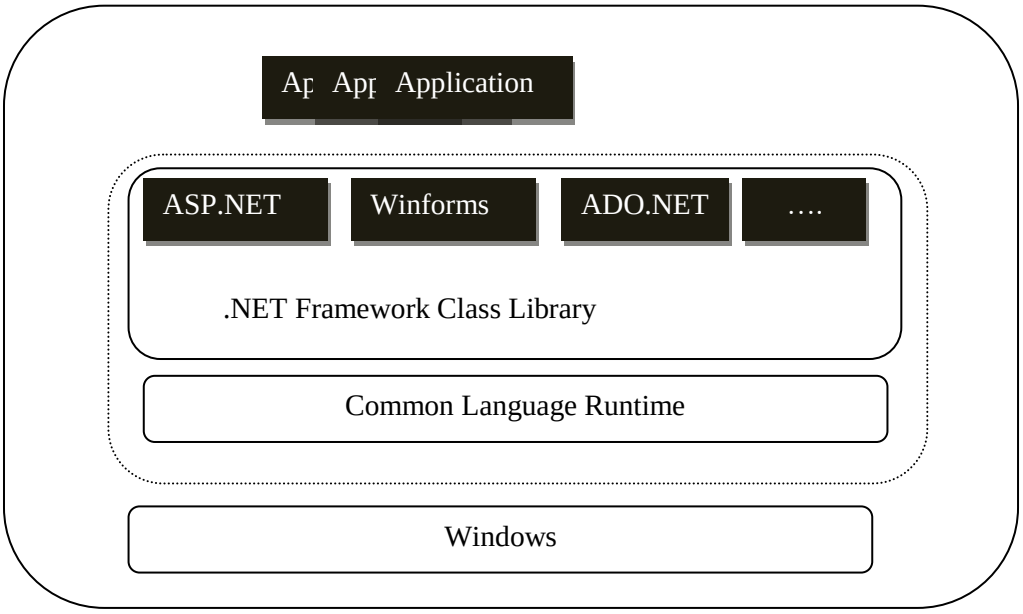


Figure 1: La plateforme Microsoft .NET

1) Le Common Intermediate Language

La plateforme .Net a cette particularité qu'elle n'est pas associée à un langage en particulier, contrairement aux autres API mises à disposition jusqu'alors (C pour Win16/13, C++ pour MFC, etc.).

En effet, la compilation d'un programme écrit dans un langage supportant la plateforme .Net génère non pas un code machine spécifique à une architecture donnée, mais un code « intermédiaire », utilisant un jeu d'instructions indépendant du processeur et qui sera exécuté par une machine virtuelle.

C'est ensuite cette machine virtuelle qui retranscrira le code intermédiaire dans un code machine correspondant à l'architecture sur laquelle est exécuté le programme.

Ce code intermédiaire porte le nom de « Common Intermediate Language » ou CIL.

Le CIL était initialement connu sous le nom de Microsoft Intermediate Language ou MSIL durant les bêtas du langage .NET. Après la standardisation du C# et de la CLI, le bytecode fut officiellement référencé sous le nom de CIL. Les utilisateurs précoces de la technologie continuent néanmoins à se référer au terme MSIL.

Compilateur JIT/NGEN

Durant la compilation .NET, le code source est transformé en un code CIL portable, indépendant de la plateforme et du processeur, et appelé bytecode.

Ce bytecode est compilé par la CLR/DLR en temps réel pour obtenir un code immédiatement exécutable par le processeur. Durant cette compilation, le compilateur (JIT) effectue un grand nombre de tâches pour éviter des accès illégaux à la mémoire :

- optimisation spécifique à la plateforme
- sécurisation des types
- vérification des assembly

Cette compilation peut aussi être réalisée avec un générateur natif d'image (NGEN). Cet outil a pour but de supprimer le temps d'attente dû à la compilation qui a lieu au niveau des CLR et DLR. Attention, l'image binaire native est placée dans le cache des 'assemblies' mais nécessite pour s'exécuter le fichier d'origine (certaines informations ne sont pas recopiées dans l'image)[1].

.NET medata

.NET enregistre les informations concernant les classes compilées dans un fichier de nom metadata. Ces données agissent comme la bibliothèque Component Object Model, et permettent aux applications compatibles de découvrir les interfaces, les classes, les types, et les méthodes présentes dans le code assembleur. Le processus de lecture de ces données est appelé réflexion. Ces données peuvent être lues en utilisant l'outil ILDASM fourni avec le SDK .NET Framework.

Toute la CIL est autodescriptive, grâce aux Métadonnées .NET. La CLR vérifie les métadonnées pour s'assurer que la bonne méthode est appelée. Les métadonnées sont généralement générées par les compilateurs des langages, mais les développeurs peuvent aussi créer leurs propres métadonnées via l'utilisation d'attributs personnalisés. Les métadonnées contiennent aussi des informations à propos des assemblages et sont aussi utilisées pour implémenter la capacité de réflexion du .NET Framework.

.NET assemblies

Le code CIL est stocké dans les assemblages .NET (ou assemblies).

L'assemblage est le bloc de structuration fondamental des applications .NET. Un assemblage regroupe l'ensemble des éléments nécessaires au bon fonctionnement d'une application (ou partie d'une application) : exécutables, métadonnées, autorisations, ressources (images...), etc. Le concept d'assemblage a été introduit pour résoudre les problèmes d'installation, d'évolution de version, d'internationalisation, de contexte d'exécution, de conflits de DLL... À ce titre, c'est une unité de déploiement indivisible.

Les assemblages sont constitués d´un ou plusieurs fichiers, dont l'un doit contenir un document XML appelé manifeste. Le manifeste[2] contient :

- la liste de l'ensemble des fichiers utilisés (exécutables, DLL, données, images, ressources)
- les métadonnées,

- la liste des autres assemblages utilisés par l'application
- l'ensemble des informations liées aux autorisations et à la sécurité de l'assemblage.

Les assemblages .NET sont enregistrés au format exécutable portable (PE) courant sur la plate-forme Windows pour tous les fichiers DLL ou EXE. Le nom complet d'un assemblage (à ne pas confondre avec le nom du fichier sur le disque) contient son nom simple, son numéro de version, sa culture et sa clé publique. La clé publique est unique et est générée à partir du hachage de l'assemblage après sa compilation. En conséquence, deux assemblages avec la même clé publique sont garantis d'être identiques. Une clé privée peut aussi être spécifiée ; elle est uniquement connue du créateur de l'assemblage et peut être utilisée pour le nommage fort de celui-ci. Cela garantit que l'assemblage est du même auteur lors de la compilation d'une nouvelle version.

Le code CIL d'un assemblage .NET existe sous deux formes : exécutables (process assemblies) et DLL (library assemblies). Lors de la compilation, le choix du format final du fichier contenant le code source ne dépend pas de l'extension du fichier mais d'une information stockée dans un fichier PE. Ce fait explique que, dans un même répertoire, deux fichiers de même nom mais avec des extensions différentes ne peuvent par défaut exister. Ce problème a été résolu par l'utilisation d'une clé publique/privée pour signer une DLL ou un exécutable et par l'introduction par .NET du GAC.

Si nous rentrons dans le principe de fonctionnement de CIL :

CIL (Langage Intermédiaire Commun) est un bytecode, c'est le langage portable de la plateforme .NET dans lequel sont compilés les codes sources écrits dans les langages de haut niveau. Son fonctionnement est basé sur une pile et il est exécuté par une machine virtuelle.*

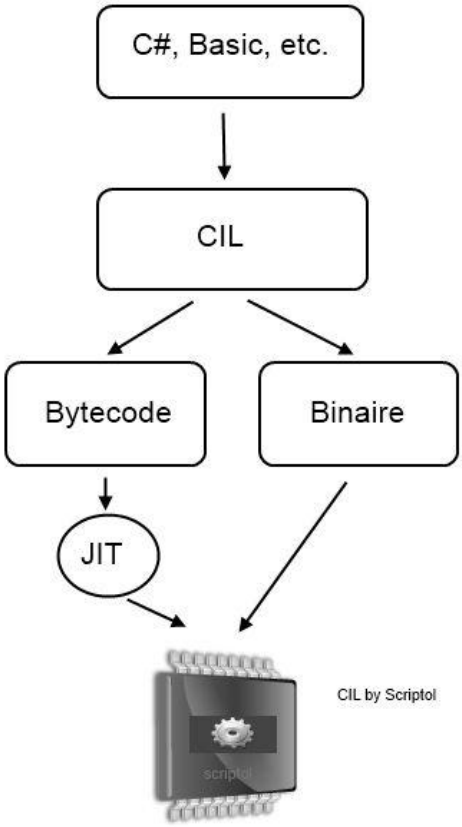
On l'appelle aussi IL ou MSIL (MicroSoft Intermediate Language) qui était son nom initial avant la standardisation de la CLI (Common Language Infrastructure) dont il fait partie et avec laquelle on ne doit pas le confondre.

Le code source de haut niveau (C#, Basic ou autre langage) est compilé en CIL et stocké dans une assembly (ou un assemblage).

L'assembly reprend le format de fichier PE (Portable Executable) qui est aussi celui des .dll et des .exe, et comprend un fichier manifeste contenant les medadata de l'assemblage, qui est l'interface du code avec les autres composants des logiciels qui l'utilisent.

Ce code est compilé en bytecode pour être interprété par une JIT, ou en binaire pour être directement exécuté par le processeur.

Le langage CIL peut effectuer des opérations arithmétiques, logiques et a des structures de contrôle (boucles, if, etc), fait des appels de fonctions et méthodes, la gestion de pile, le chargement et sauvegarde de données, une conversion de types (nombre en chaîne...), supporte les exceptions et la concurrence.



2) Le Common Language Runtime (CLR)

Il s'agit de l'environnement dans lequel s'exécute tout code managé. Il fournit de nombreux services, que ce soit la gestion de la mémoire (ramasse-miette), la gestion des exceptions, la gestion des tâches, le chargement de code managé, les types de base (entier, chaîne de caractères…).

Le CLR est souvent vu comme une machine virtuelle, dans la mesure où il permet l'exécution d'un code écrit pour une architecture différente.

Il est à noter que le CLR n'est pas une spécification de l'environnement dans lequel s'exécute un code managé, mais une implémentation, réalisée par Microsoft, de la spécification « Common Language Infrastructure » (CLI). C'est cette spécification qui est LA référence décrivant l'environnement d'exécution.

On peut noter le CLR est considéré comme l'implémentation de référence du CLI, au point qu'il est parfois fait mention de CLR au lieu de CLI. Par exemple, le projet Mono, dont l'objectif est de fournir un environnement .Net sur des plateformes comme Linux ou MacOS, se décrit comme une implémentation open source du Framework .Net de Microsoft basé sur le standard ECMA pour C# et le « Common Language Runtime ».

3) Chargement du Common Language Runtime

Pour savoir si Microsoft .NET Framework est installé, il faut chercher le fichier MSCorEE.dll dans %SystemRoot%\System32.

Le CLR est livré en standard avec les versions de Windows dans :

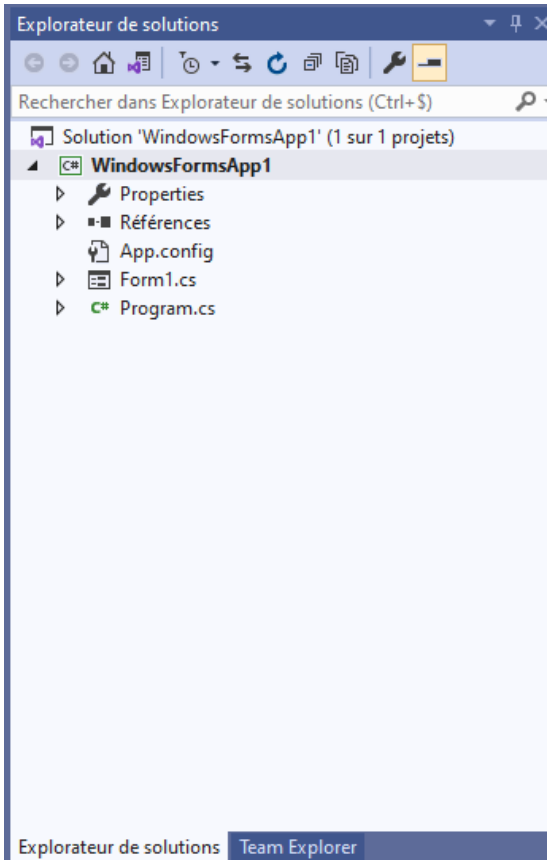
- %SystemRoot%\Microsoft.NET\Framework;
- %SystemRoot%\Microsoft.NET\Framework64;
- Pour obtenir le numéro de version du Framework, il faut regarder la version du fichier mscorlib.dll. Exemple : version 4.8.4200.0 pour le Framework 4.8.

Il existe plusieurs plateforme pour faire tourner .NET:

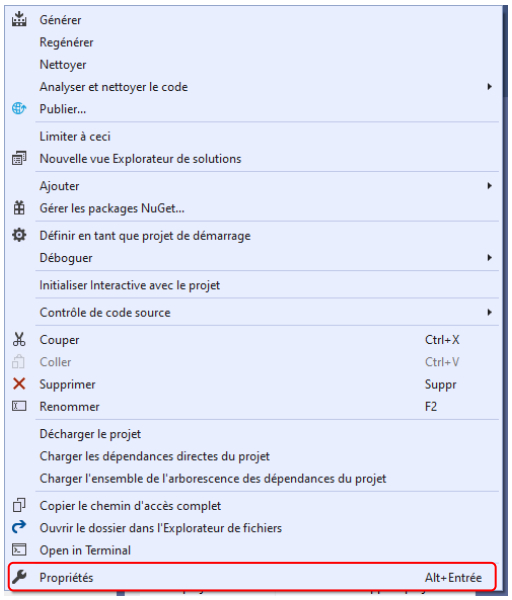
- Architecture x86 (Windows 32-bit);
- Architecture x64 (Windows 64-bit);
- Architecture ARM (Windows RT).

Il est possible de faire tourner son code sur une architecture spécifique si l'on fait des accès unsafe à des DLL systèmes spécifiques mais généralement, on execute son code en mode anycpu.

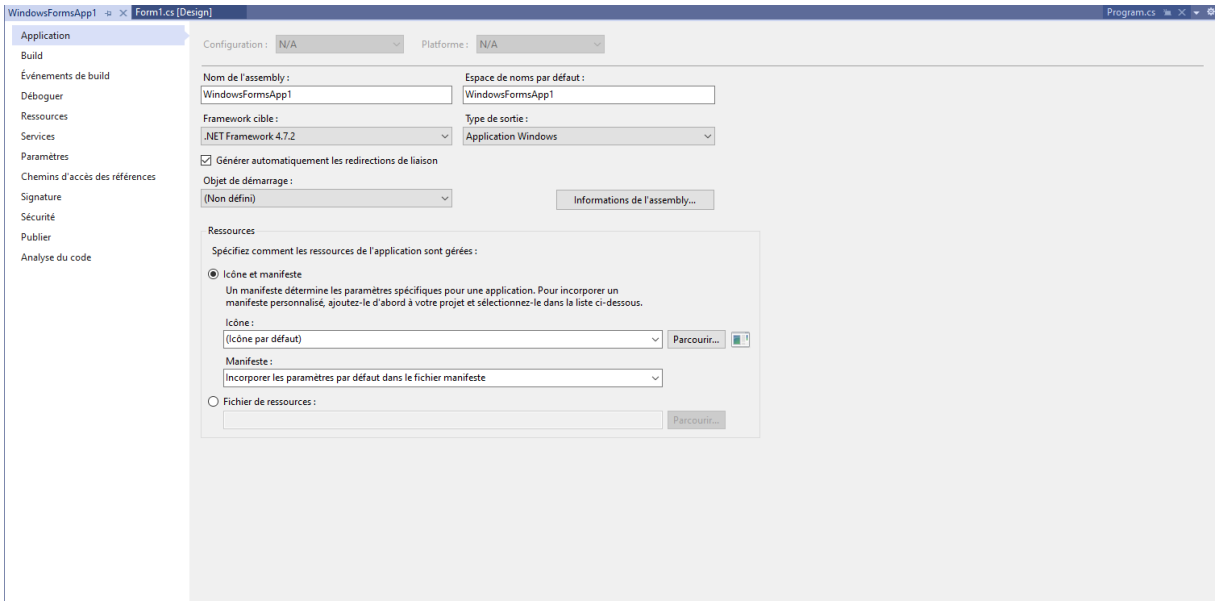
Prenons comme exemple un projet Winform :



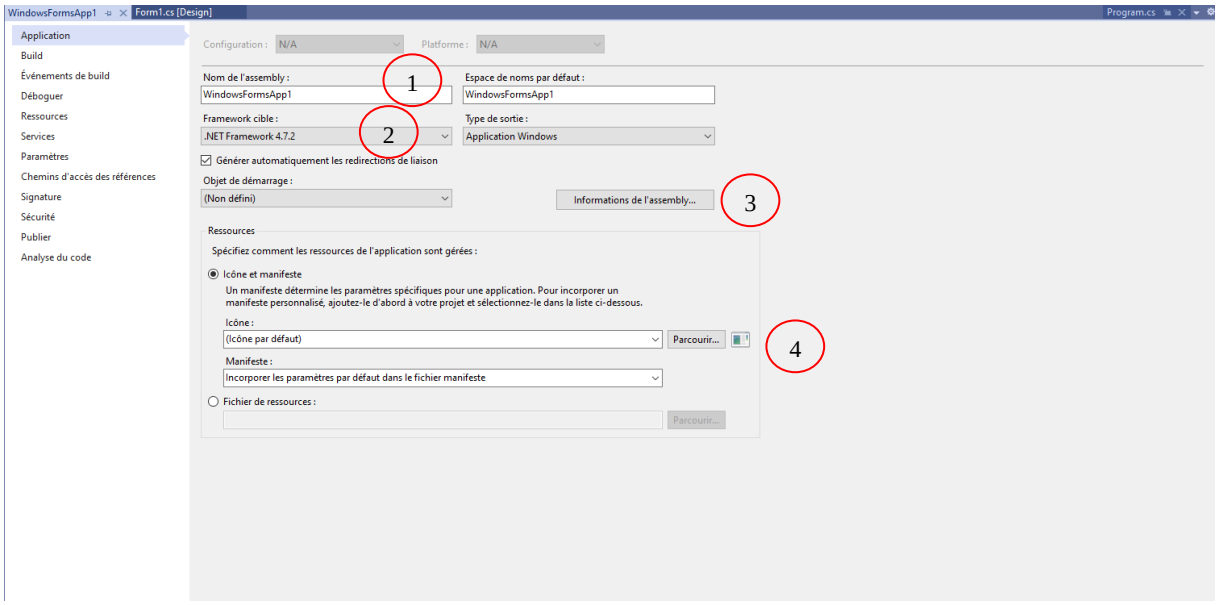
Cliquez sur WindowsFormsApp1 clique droite de la souris, puis sur propriété



Sur propriétés, on obtient la fenêtre suivante :



Dans l'onglet « Application » sont les paramètres de l'application.



Descriptions :

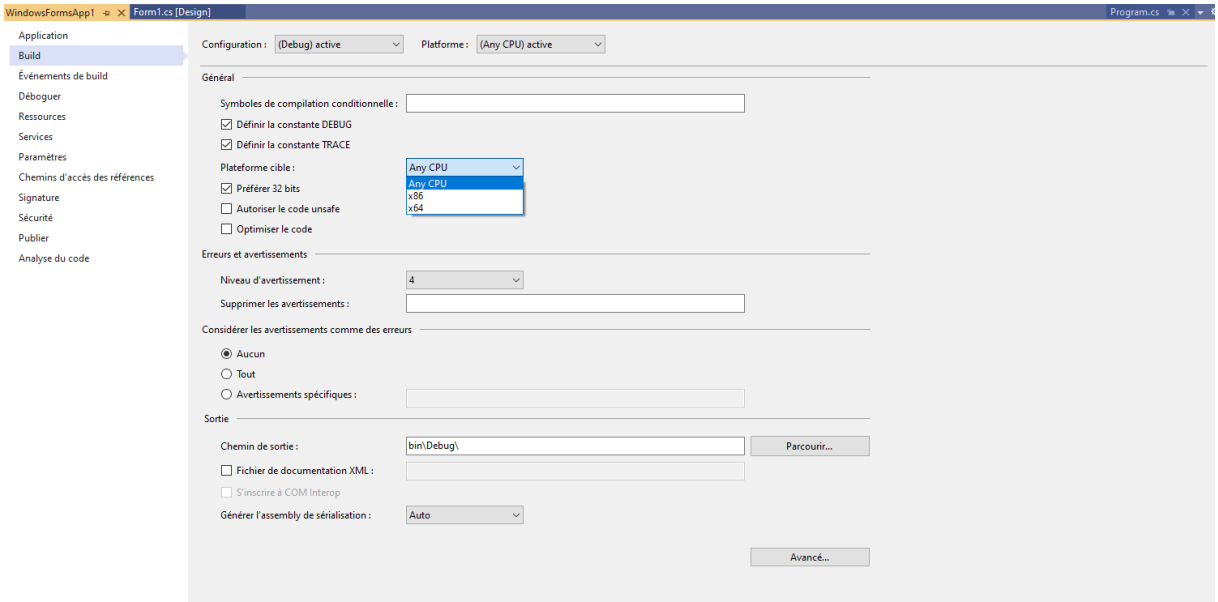
En (1) est le nom de l'assembly (assembleur) ;

En (2) est le choix du Framework ;

En (3) sont les informations de l'assembly ;

En (4) est l'ajout de l'icône et d'un manifeste.

Nous passons maintenant au deuxième onglet le « Build » :



Nous pouvons remarquer qu'il est possible de changer la cible du compilateur.

En effet différents compilations sont possibles dans un logiciel sous Visual Studio avec les Framework .NET.

Je les résume dans ce tableau :

/plateform Switch	Module	Windows x86	Windows x64
Anycpu	PE32	App 32-bit	App 64-bit
anycpu32bitprefered	PE32	App 32-bit	App 64-bit
x86	PE32/x86	App 32-bit	App WoW64
x64	PE32+/x64	Ne tourne pas	App 64-bit
ARM	PE32/ARM	Ne tourne pas	Ne tourne pas

WoW64 (Windows on Windows 64) permet de faire tourner un module 32-bit dans un espace mémoire de processus 64-bit.

Windows examine l'entête du fichier EXE pour savoir s'il faut créer un processus 32-bit ou 64-bit et charge la version de MSCorEE.dll en version x86, x64 ou ARM dans l'espace mémoire du processus. Ensuite, le thread principal du processus appelle une routine dans MSCorEE.dll qui initialise le CLR, charge l'assembly EXE et appelle le point d'entrée Main. A ce stade, l'application managée est up et tourne.

4) Exécution de code de l'assembly

Les assemblies contiennent des métadonnées et du code IL (alias MSIL). IL est un langage machine indépendant du CPU. C'est un langage de plus haut niveau que les assembleurs traditionnels et il peut être considéré comme un assembleur oriéntée objet: il peut appeler des methods virtuelles, lancer des exceptions, etc.

Il est possible d'écrire du code IL directement. Il existe un compilateur IL nommé ILAsm.exe et un désassembleur ILDAsm.exe. N'oubliez pas qu'un compilateur de la plateforme .NET ne fait que traduire le code d'un langage en IL, et que c'est code IL qui permet d'accéder aux services offerts par le CLR.

Ce qui est important de savoir qu'il est possible de mixer les languages dans une solution. Vous pouvez faire une DLL en VB.NET, une autre en C#, une autre en F# et les reference dans EXE fait en C#. C'est une possiblité fantastique pour exploiter les capacities de tous les langages.

Pour executer une méthode, le code IL doit être converti en instructions CPU. C'est le job du compilateur JIT (just-in-time compiler).

4.1 Le Dynamic Language Runtime (CLR)

Le **Dynamic Language Runtime (DLR)** est une surcouche facilitant l'implémentation et l'interopérabilité des langages dynamiques. Le DLR permet de supporter plusieurs langages. Cette technologie est utilisée par Microsoft pour la plateforme .NET et Oracle pour le langage Java.

Le CLR que nous venons de voir ne supporte que les langages statiquement typés, c'est-à-dire que le type d'un objet doit être connu au moment de la compilation.

Le CLR de Microsoft est l'équivalent de la machine virtuelle Java (JVM). Le CLR ne gère que les langages propres à Microsoft tandis que le DLR de Microsoft a été créé pour prendre en charge d'autres langages. Oracle suit la même politique d'ouverture en ajoutant à sa JVM un DLR nommé Da Vinci Machine permettant la gestion d'autres langages que Java. Le terme JVM devient donc inapproprié. Des langages tiers avaient déjà été portés sur la JVM en reprenant la syntaxe de langages existants. La raison principale des DLR est de faciliter le portage de langages tiers ; il est en effet très difficile de faire fonctionner un code dynamique (Python, Ruby) sur un environnement qui est typé lors de la compilation. Le portage d'un langage sur le DLR est donc plus facile que sur le CLR.

4.2 Le Common Type System (CTS)

Comme précisé précédemment, la plateforme .Net ne dépend pas d'un langage particulier. Cela signifie d'une part que tout développeur peut écrire son code dans tout langage qui peut être compilé en CIL, mais également qu'il est nécessaire que des codes écrits dans des langages différents puissent interagir.

Et c'est justement là qu'intervient le « Common Type System » en décrivant tous les types supportés (entier, chaîne de caractères, etc.) d'une manière indépendante des langages afin qu'ils puissent interagir entre eux de manière transparente. Ainsi, un code C# pourra utiliser une classe définie dans une autre bibliothèque, qu'importe qu'elle ait été écrite en C#, en VB.Net, en C++/CLI, etc.

Pour parvenir à cela, le « Common Type System » :

- définit le cadre général afin de permettre de passer d'un langage à un autre ;
- fournit un modèle orienté-objet afin de supporter l'implémentation de nombreux langages sur la plateforme .Net ;
- définit un ensemble de règles que les langages doivent suivre ;
- fournit une liste de types primitifs (entier, booléen, chaîne de caractères, etc.).

Le CTS définit le type Object et les types Value et References :

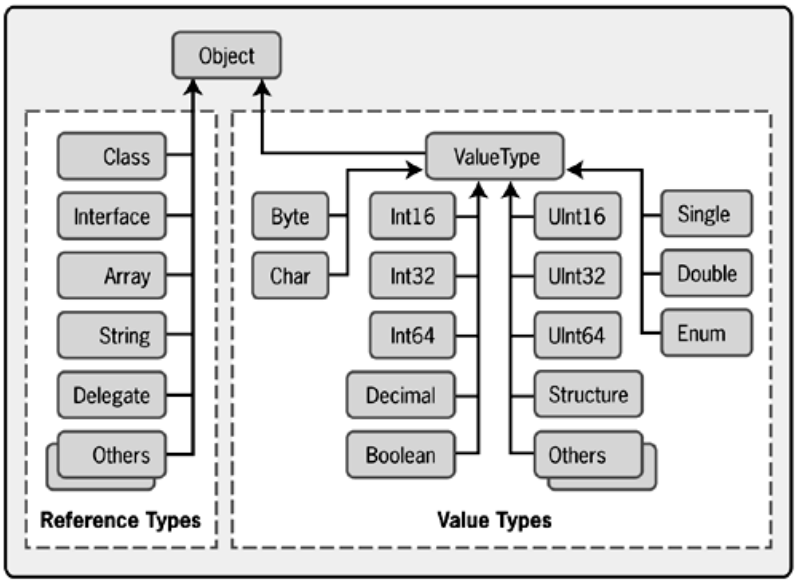


Figure 2: Le CTS

4.3 les Types . NET

Tous les types dans .NET sont soit des types valeur, soit des types référence.

Les *types valeur* sont des types de données dont les objets sont représentés par la valeur réelle de l'objet. Si une instance d'un type valeur est assignée à une variable, cette variable reçoit une copie actualisée de la valeur.

Les *types référence* sont des types de données dont les objets sont représentés par une référence (identique à un pointeur) à la valeur réelle de l'objet. Si un type référence est assigné à une variable, celle-ci référence (ou pointe vers) la valeur d'origine. Aucune copie n'est effectuée.

Le système de type commun (CTS, Common Type System) dans .NET prend en charge les cinq catégories de types suivantes :

- Classes
- Structures
- Énumérations
- Interfaces
- Délégués

4.3.1 CLASSES

Une classe est un type référence qui peut être dérivé directement d'une autre classe et qui est implicitement dérivé de *System.Object*. Une classe définit les opérations qu'un objet (qui est une instance de la classe) peut effectuer (méthodes, événements ou propriétés) et les données que l'objet contient (champs). Bien qu'une classe inclue généralement la définition et l'implémentation (contrairement aux interfaces, par exemple, qui contiennent uniquement la définition sans l'implémentation), elle peut comporter un ou plusieurs membres dépourvus d'implémentation.

Le tableau suivant décrit certaines des caractéristiques qu'une classe peut avoir. Chaque langage prenant en charge le runtime fournit un moyen d'indiquer qu'une classe ou qu'un membre d'une classe a une ou plusieurs de ces caractéristiques. Cependant, il se peut que des langages de programmation individuels qui ciblent .NET ne mettent pas toutes ces caractéristiques à disposition.

Caractéristique	Description
sealed	Spécifie qu'une autre classe ne peut pas être dérivée de ce type.
implémente	Indique que la classe utilise une ou plusieurs interfaces en fournissant des implémentations des membres d'interface.
abstract	Indique que la classe ne peut pas être instanciée. Pour l'utiliser, vous devez dériver une autre classe de celle-ci.
hérite	Indique que des instances de la classe peuvent être utilisées partout où la classe de base est spécifiée. Une classe dérivée qui hérite d'une classe de base peut utiliser l'implémentation de n'importe quel membre public fourni par la classe de base, ou la classe dérivée peut remplacer l'implémentation des membres publics par sa propre implémentation.
exported ou not exported	Indique si une classe est visible à l'extérieur de l'assembly dans lequel elle est définie. Cette caractéristique s'applique uniquement aux classes de niveau supérieur, et pas aux classes imbriquées.

Les membres de classe sans implémentation sont des membres abstraits. Une classe qui possède un ou plusieurs membres abstraits est elle-même abstraite ; il n'est pas possible d'en créer de nouvelles instances. Certains langages qui ciblent le runtime vous permettent de marquer une classe comme abstraite même si aucun de ses membres n'est abstrait. Vous pouvez utiliser une classe abstraite lorsque vous voulez encapsuler un ensemble de fonctionnalités de base dont des classes dérivées peuvent hériter ou qu'elles peuvent substituer lorsque cela est approprié. Les classes qui ne sont pas abstraites sont qualifiées de classes concrètes.

Une classe peut implémenter n'importe quel nombre d'interfaces, mais elle ne peut hériter que d'une seule classe de base en plus de la classe *System.Object*, de laquelle toutes les classes héritent implicitement. Toutes les classes doivent avoir au moins un constructeur qui initialise de nouvelles instances de la classe. Si vous ne définissez pas explicitement un constructeur, la plupart des compilateurs fournissent automatiquement un constructeur sans paramètre.

4.3.2STRUCTURES

Une structure est un type valeur qui dérive implicitement de *System.ValueType* qui, à son tour, est dérivé de *System.Object*. Une structure est utile pour représenter des valeurs dont les besoins en mémoire sont faibles et pour passer des valeurs en tant que paramètres par valeur à des méthodes qui ont des paramètres fortement typés. Dans .NET, tous les types de données primitifs (Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32 et UInt64) sont définis en tant que structures.

Comme les classes, les structures définissent à la fois les données (les champs de la structure) et les opérations qui peuvent être exécutées sur ces données (les méthodes de la structure). Cela signifie que vous pouvez appeler des méthodes sur des structures, notamment les méthodes virtuelles définies sur les classes System.Object et System.ValueType, ainsi que toute méthode définie sur le type valeur lui-même. En d'autres termes, les structures peuvent comporter des champs, des propriétés et des événements, ainsi que des méthodes statiques et non statiques. Vous pouvez créer des instances de structures, les passer en tant que paramètres, les stocker en tant que variables locales ou les stocker dans un champ d'un autre type valeur ou type référence. Les structures peuvent aussi implémenter des interfaces.

Les types valeur diffèrent également des classes par plusieurs aspects. Tout d'abord, bien qu'ils héritent implicitement de *System.ValueType*, ils ne peuvent pas hériter directement de n'importe quel type.

De même, tous les types valeur sont sealed, ce qui signifie qu'aucun autre type ne peut en être dérivé. De plus, ils ne nécessitent pas de constructeurs.

Pour chaque type valeur, le Common Language Runtime fournit un type boxed correspondant qui est une classe ayant le même état et le même comportement que le type valeur. Une instance d'un type valeur est boxed lorsqu'elle est passée à une méthode qui accepte un paramètre de type *System.Object*. Elle est unboxed (c'est-à-dire reconvertie d'une instance d'une classe en instance d'un type valeur) lorsque le contrôle retourne d'un appel de méthode qui accepte un type valeur comme paramètre par référence. Certains langages imposent l'utilisation d'une syntaxe spéciale lorsque le type boxed est requis ; d'autres utilisent automatiquement le type boxed lorsqu'il est nécessaire. Lorsque vous définissez un type valeur, vous définissez le type boxed et le type unboxed.

4.3.3 Enumérations

Une énumération est un type valeur qui hérite directement de *System.Enum* et qui fournit des noms de remplacement pour les valeurs d'un type primitif sous-jacent. Un type énumération a un nom, un type sous-jacent qui doit être l'un des types d'entiers signés ou non signés intégrés (tels que Byte, Int32 ou UInt64) et un ensemble de champs. Les champs sont des champs statiques de type Literal, chacun représentant une constante. La même valeur peut être assignée à plusieurs champs. Dans ce cas, vous devez marquer l'une des valeurs comme valeur d'énumération primaire pour la réflexion et la conversion de chaînes.

Vous pouvez assigner une valeur du type sous-jacent à une énumération et vice-versa (aucun cast n'est requis par le runtime). Vous pouvez créer une instance d'une énumération et appeler les méthodes de *System.Enum*, ainsi que toute méthode définie sur le type sous-jacent de l'énumération. Cependant, il est possible que certains langages ne vous permettent pas de passer une énumération en tant que paramètre lorsqu'une instance du type sous-jacent est requise (ou vice versa).

Les restrictions supplémentaires suivantes s'appliquent aux énumérations :

- Elles ne peuvent pas définir leurs propres méthodes.
- Elles ne peuvent pas implémenter d'interfaces.
- Elles ne peuvent pas définir des propriétés ou des événements.
- Elles ne peuvent pas être génériques, à moins qu'elles le soient uniquement parce qu'elles sont imbriquées dans un type générique. Par conséquent, une énumération ne peut pas avoir de paramètres de type propres.

L'attribut FlagsAttribute[3] désigne un genre particulier d'énumération appelé « champ de bits ». Le runtime lui-même ne fait pas de distinction entre les énumérations traditionnelles et les champs de bits, mais votre langage pourrait le faire. Lorsque cette distinction est effectuée, les opérateurs binaires peuvent être utilisés sur les champs de bits, mais pas sur les énumérations, pour générer des valeurs sans nom. Les énumérations sont généralement utilisées pour des listes d'éléments uniques, tels que les jours de la semaine, des noms de pays ou de région, etc. Les champs de bits sont généralement utilisés pour des listes de qualités ou de quantités pouvant être utilisées en combinaison, telle que `Red And Big And Fast`.

Exemple :

```
using System;
using System.Collections.Generic;
// A traditional enumeration of some root vegetables.
```

[3] Indique qu'une énumération peut être traitée comme un champ de bits, c'est-à-dire un ensemble d'indicateurs.

```
public enum SomeRootVegetables
{
   HorseRadish,
   Radish,
   Turnip
}

// A bit field or flag enumeration of harvesting seasons.
[Flags]
public enum Seasons
{
   None = 0,
   Summer = 1,
   Autumn = 2,
   Winter = 4,
   Spring = 8,
   All = Summer | Autumn | Winter | Spring
}

public class Example
{
  public static void Main()
  {
     // Hash table of when vegetables are available.
     Dictionary<SomeRootVegetables, Seasons> AvailableIn = new
Dictionary<SomeRootVegetables, Seasons>();

     AvailableIn[SomeRootVegetables.HorseRadish] = Seasons.All;
     AvailableIn[SomeRootVegetables.Radish] = Seasons.Spring;
     AvailableIn[SomeRootVegetables.Turnip] = Seasons.Spring |
        Seasons.Autumn;

     // Array of the seasons, using the enumeration.
     Seasons[] theSeasons = new Seasons[] { Seasons.Summer, Seasons.Autumn,
        Seasons.Winter, Seasons.Spring };

     // Print information of what vegetables are available each season.
     foreach (Seasons season in theSeasons)
     {
       Console.Write(String.Format(
          "The following root vegetables are harvested in {0}:\n",
          season.ToString("G")));
       foreach (KeyValuePair<SomeRootVegetables, Seasons> item in AvailableIn)
       {
         // A bitwise comparison.
         if (((Seasons)item.Value & season) > 0)
            Console.Write(String.Format("  {0:G}\n",
               (SomeRootVegetables)item.Key));
       }
     }
  }
}
// The example displays the following output:
//    The following root vegetables are harvested in Summer:
//      HorseRadish
```

```
//    The following root vegetables are harvested in Autumn:
//      Turnip
//      HorseRadish
//    The following root vegetables are harvested in Winter:
//      HorseRadish
//    The following root vegetables are harvested in Spring:
//      Turnip
//      Radish
//      HorseRadish
```

4.3.4Interfaces

Une interface définit un contrat qui spécifie une relation « peut faire » ou « a un ». Les interfaces permettent souvent d'implémenter des fonctionnalités, comme la comparaison et le tri (interfaces IComparable et IComparable<T>), le test pour l'égalité (interface IEquatable<T>) ou l'énumération d'éléments dans une collection (interfaces IEnumerable et IEnumerable<T>). Les interfaces peuvent avoir des propriétés, des méthodes et des événements qui sont tous des membres abstraits, c'est-à-dire que bien que l'interface définisse les membres et leurs signatures, elle laisse le type qui implémente l'interface définir la fonctionnalité de chaque membre d'interface. En d'autres termes, toute classe ou structure qui implémente une interface doit fournir des définitions pour les membres abstraits déclarés dans l'interface. Une interface peut imposer à une classe ou à une structure d'implémentation d'implémenter une ou plusieurs autres interfaces.

Les restrictions suivantes s'appliquent aux interfaces :

- Une interface peut être déclarée avec n'importe quelle accessibilité, mais les membres d'interface doivent tous avoir une accessibilité publique.
- Les interfaces ne peuvent pas définir de constructeurs.
- Les interfaces ne peuvent pas définir de champs.
- Les interfaces peuvent définir uniquement des membres d'instance. Elles ne peuvent pas définir des membres statiques.

Chaque langage doit fournir des règles de mappage d'une implémentation à l'interface qui nécessite le membre, car plusieurs interfaces peuvent déclarer un membre avec la même signature, et ces membres peuvent avoir des implémentations séparées.

4.3.5 Délégués

Les délégués sont des types référence qui jouent un rôle similaire à celui des pointeurs fonction en C++. Ils sont utilisés pour les gestionnaires d'événements et les fonctions de rappel dans .NET. Contrairement aux pointeurs fonction, les délégués sont de type sécurisé, sûr et vérifiable. Un type délégué peut représenter n'importe quelle méthode d'instance ou méthode statique ayant une signature compatible.

Le paramètre d'un délégué est compatible avec le paramètre correspondant d'une méthode si le type de paramètre du délégué est plus restrictif que le type de paramètre de la méthode. En effet, cela garantit qu'un argument transmis au délégué peut être transmis à la méthode en toute sécurité.

De même, le type de retour d'un délégué est compatible avec le type de retour d'une méthode si le type de retour de la méthode est plus restrictif que le type de retour du délégué, car cela garantit que le cast de la valeur de retour de la méthode peut être effectué sans risque au type de retour du délégué.

Par exemple, un délégué ayant un paramètre de type IEnumerable et un type de retour Object peut représenter une méthode ayant un paramètre de type Object et une valeur de retour de type IEnumerable. Pour obtenir des informations supplémentaires ainsi qu'un code d'exemple, consultez Delegate.CreateDelegate(Type, Object, MethodInfo).

On dit qu'un délégué est lié à la méthode qu'il représente. Outre cette liaison, un délégué peut être lié à un objet. L'objet représente le premier paramètre de la méthode, et est passé à cette dernière chaque fois que le délégué est appelé. Si la méthode est une méthode d'instance, l'objet dépendant est passé en tant que paramètre `this` implicite (`Me` en Visual Basic) ; si la méthode est statique, l'objet est passé en tant que premier paramètre formel de la méthode, et la signature du délégué doit correspondre aux paramètres restants. Pour obtenir des informations supplémentaires ainsi qu'un code d'exemple, consultez System.Delegate.

Tous les délégués héritent de System.MulticastDelegate, qui hérite de System.Delegate. Les langages C#, Visual Basic et C++ n'autorisent pas l'héritage à partir de ces types. Ils fournissent à la place des mots clés pour la déclaration des délégués.

Étant donné que les délégués héritent de MulticastDelegate, un délégué dispose d'une liste d'appel, liste des méthodes représentées par le délégué et exécutées lorsque celui-ci est appelé. Toutes les méthodes de la liste reçoivent les arguments fournis lorsque le délégué est appelé.

4.4 Définitions de type

Une définition de type inclut les éléments suivants :

- les attributs définis sur le type ;
- L'accessibilité du type (visibilité).
- le nom du type ;
- le type de base du type ;
- les interfaces implémentées par le type ;
- les définitions de chacun des membres du type.

4.4.1 Attributs

Les attributs fournissent des métadonnées définies par l'utilisateur supplémentaires. La plupart du temps, ils sont utilisés pour stocker les informations supplémentaires relatives à un type de l'assembly, ou pour modifier le comportement d'un membre de type dans l'environnement au moment du design ou dans l'environnement d'exécution.

Les attributs sont des classes qui héritent de *System.Attribute*. Les langages qui prennent en charge l'utilisation d'attributs ont chacun leur propre syntaxe pour l'application d'attributs à un élément du langage. Les attributs peuvent être appliqués à presque n'importe quel élément de langage ; les éléments spécifiques auxquels un attribut peut être appliqué sont définis par l'AttributeUsageAttribute qui est appliqué à cette classe d'attributs.

4.4.2 Accessibilité des types

Tous les types ont un modificateur qui régit leur accessibilité à partir d'autres types. Le tableau suivant décrit les accessibilités de type prises en charge par le runtime.

Accessibilité	Description
public	Le type est accessible par tous les assemblys.
assembly	Le type est accessible uniquement à partir de cet assembly.

L'accessibilité d'un type imbriqué dépend de son domaine d'accessibilité, qui est déterminé par l'accessibilité déclarée du membre et le domaine d'accessibilité du type conteneur immédiat. Toutefois, le domaine d'accessibilité d'un type imbriqué ne peut pas dépasser celui du type conteneur.

Le domaine d'accessibilité d'un membre imbriqué M déclaré dans un type T à l'intérieur d'un programme P est défini de la manière suivante (en notant que M pourrait lui-même être un type) :

- Si l'accessibilité déclarée de M est public, le domaine d'accessibilité de M correspond à celui de T.
- Si l'accessibilité déclarée de M est protected internal, le domaine d'accessibilité de M correspond à l'intersection du domaine d'accessibilité de T avec le texte de programme de P et le texte de programme de n'importe quel type dérivé de T déclaré en dehors de P.
- Si l'accessibilité déclarée de M est protected, le domaine d'accessibilité de M correspond à l'intersection du domaine d'accessibilité de T avec le texte de programme de T et n'importe quel type dérivé de T.
- Si l'accessibilité déclarée de M est internal, le domaine d'accessibilité de M correspond à l'intersection du domaine d'accessibilité de T avec le texte de programme de P.
- Si l'accessibilité déclarée de M est private, le domaine d'accessibilité de M correspond au texte de programme de T.

4.4.3 Noms de types

Le système de type commun (CTS, Common Type System) impose uniquement deux restrictions sur les noms :

- Tous les noms sont encodés comme des chaînes de caractères Unicode (16 bits).
- Les noms ne peuvent pas incorporer la valeur (16 bits) 0x0000.

Toutefois, la plupart des langages imposent des restrictions supplémentaires sur les noms de types. Toutes les comparaisons sont effectuées octet par octet ; elles respectent donc la casse et sont indépendantes des paramètres régionaux.

Bien qu'un type puisse référencer des types d'autres modules et assemblys, il doit être entièrement défini à l'intérieur d'un seul module .NET. (En fonction de la prise en charge du compilateur, toutefois, il peut être divisé en plusieurs fichiers de code source.) Les noms de types doivent être uniques uniquement dans un espace de noms. Pour identifier complètement un type, le nom du type doit être qualifié par l'espace de noms qui contient l'implémentation du type.

4.4.4 Membres de type

Le runtime vous permet de définir les membres de votre type, ce qui spécifie le comportement et l'état d'un type. Les membres de type incluent les éléments suivants :

- un champs (fields)
- Propriétés (property)
- Méthodes (methods)
- Constructeurs (build)
- Événements (event)
- Types imbriqués

4.4.4.1 Champs

Un champ décrit et contient une partie de l'état du type. Les champs peuvent correspondre à n'importe quel type pris en charge par le runtime. La plupart du temps, les champs sont soit `private`, soit `protected`, de sorte qu'ils sont accessibles uniquement à partir de la classe ou d'une classe dérivée. Si la valeur d'un champ peut être modifiée en dehors de son type, un accesseur set de propriété est en général utilisé. Les champs exposés publiquement sont généralement en lecture seule et peuvent être de deux types :

- Constantes, dont la valeur est assignée au moment du design. Ce sont des membres statiques d'une classe, bien qu'ils ne soient pas définis à l'aide du mot clé `static` (`Shared` en Visual Basic).
- Des variables en lecture seule, dont les valeurs peuvent être assignées dans le constructeur de classe.

L'exemple ci-dessous illustre ces deux utilisations de champs en lecture seule.

```
using System;

public class Constants
{
   public const double Pi = 3.1416;
   public readonly DateTime BirthDate;

   public Constants(DateTime birthDate)
   {
      this.BirthDate = birthDate;
   }
}

public class Example
{
   public static void Main()
   {
      Constants con = new Constants(new DateTime(1974, 8, 18));
      Console.Write(Constants.Pi + "\n");
      Console.Write(con.BirthDate.ToString("d") + "\n");
   }
}
// The example displays the following output if run on a system whose
current
// culture is en-US:
//    3.1416
//    8/18/1974
```

4.4.4.2 Propriétés

Une propriété nomme une valeur ou un état du type et définit des méthodes pour obtenir ou définir la valeur de la propriété. Les propriétés peuvent être des types primitifs, des collections de types primitifs, des types définis par l'utilisateur ou des collections de types définis par l'utilisateur. Les propriétés sont souvent utilisées pour maintenir l'interface publique d'un type indépendante de la représentation réelle du type. Cela permet aux propriétés de refléter des valeurs qui ne sont pas directement stockées dans la classe (par exemple, lorsqu'une propriété retourne une valeur calculée) ou d'effectuer une validation avant que des valeurs soient assignées à des champs privés. L'exemple suivant illustre ce dernier modèle.

```
using System;

public class Person
{
   private int m_Age;

   public int Age
   {
      get { return m_Age; }
      set {
         if (value < 0 || value > 125)
         {
            throw new ArgumentOutOfRangeException("The value of the Age
property must be between 0 and 125.");
         }
         else
         {
            m_Age = value;
         }
      }
   }
}
```

En plus d'inclure la propriété elle-même, le langage MSIL (Microsoft Intermediate Language) pour un type qui contient une propriété lisible inclut une `get_` méthode *PropertyName* , et le MSIL pour un type qui contient une propriété accessible en écriture inclut une `set_` méthode *PropertyName.*

4.4.4.3 Méthodes

Une méthode décrit les opérations qui sont disponibles sur le type. La signature d'une méthode spécifie les types autorisés de tous ses paramètres et de sa valeur de retour.

Bien que la plupart des méthodes définissent le nombre précis de paramètres requis pour les appels de méthode, certaines méthodes acceptent un nombre variable de paramètres. Le dernier paramètre déclaré de ces méthodes est marqué avec l'attribut ParamArrayAttribute. Les compilateurs de langage fournissent généralement un mot clé, tel que `params` en C# et `ParamArray` en Visual Basic, qui rendent l'utilisation explicite de ParamArrayAttribute inutile.

4.4.4.4 Constructeurs

Un constructeur est un genre de méthode particulier qui crée de nouvelles instances d'une classe ou d'une structure. Comme n'importe quelle autre méthode, un constructeur peut inclure des paramètres ; toutefois, les constructeurs n'ont pas de valeur de retour (en d'autres termes, ils retournent `void`).

Si le code source d'une classe ne définit pas explicitement un constructeur, le compilateur inclut un constructeur sans paramètre. Toutefois, si le code source d'une classe définit uniquement des constructeurs paramétrables, les compilateurs Visual Basic et C# ne génèrent pas de constructeur sans paramètre.

Si le code source d'une structure définit des constructeurs, ceux-ci doivent être paramétrés ; une structure ne peut pas définir un constructeur sans paramètre et les compilateurs ne génèrent pas de constructeur sans paramètre pour les structures ou autres types valeur. Tous les types valeurs disposent d'un constructeur sans paramètre implicite. Ce constructeur est implémenté par le Common Language Runtime et initialise tous les champs de la structure avec leurs valeurs par défaut.

4.5 Caractéristiques des members de type

Le système de type commun (CTS, Common Type System) permet aux membres de type d'avoir diverses caractéristiques ; toutefois, les langages ne doivent pas obligatoirement prendre en charge toutes ces caractéristiques. Le tableau suivant décrit les caractéristiques des membres.

Caractéristique	Applicable à	Description
abstract	Méthodes, propriétés et événements	Le type n'assure pas l'implémentation de la méthode. Les types qui héritent de méthodes abstraites ou qui en implémentent doivent fournir une implémentation pour la méthode. Une exception : le type dérivé est lui-même un type abstrait. Toutes les méthodes abstraites sont virtuelles.
private, family, assembly, family et assembly, family ou assembly, ou public	Tous	Définit l'accessibilité du membre : private Accessible uniquement à partir du même type que le membre ou à l'intérieur d'un type imbriqué. famille Accessible à partir du même type que le membre et à partir des types dérivés qui en héritent. assembly Accessible uniquement dans l'assembly dans lequel le type est défini. family et assembly Accessible uniquement à partir des types qui se qualifient pour un accès family et assembly. family ou assembly Accessible uniquement à partir des types qui se qualifient pour un accès family ou assembly. public Accessible à partir de n'importe quel type.

Caractéristique	Applicable à	Description
final	Méthodes, propriétés et événements	La méthode virtuelle ne peut pas être substituée dans un type dérivé.
initialize-only	Champs	La valeur peut seulement être initialisée et ne peut pas être écrite après initialisation.
instance	Champs, méthodes, propriétés et événements	Si un membre n'est pas marqué comme `static` (C# et C++), `Shared` (Visual Basic), `virtual` (C# et C++) ou `Overridable` (Visual Basic), il est membre d'instance (il n'y a pas de mot clé d'instance). La mémoire comptera autant de copies de tels membres que d'objets qui les utilisent.
littéral	Champs	La valeur assignée au champ est une valeur fixe, connue au moment de la compilation, d'un type valeur intégré. Les champs de type Literal sont parfois qualifiés de constantes.
newslot ou override	Tous	Définit la façon dont le membre interagit avec des membres hérités ayant la même signature : newslot Masque les membres hérités ayant la même signature. override Remplace la définition d'une méthode virtuelle héritée. La valeur par défaut est newslot.
static	Champs, méthodes, propriétés et événements	Le membre appartient au type sur lequel il est défini, et non à une instance particulière du type ; le membre existe même si une instance du type n'est pas créée, et il est partagé entre toutes les instances du type.
virtual	Méthodes, propriétés et événements	La méthode peut être implémentée par un type dérivé et peut être appelée de manière statique ou dynamique. Si un appel dynamique est utilisé, le type de l'instance qui effectue l'appel au moment de l'exécution détermine quelle implémentation de la méthode est appelée (plutôt que le type connu au moment de la compilation). Pour appeler une méthode virtuelle de manière statique, un cast en un type qui utilise la version désirée de la méthode devra éventuellement être effectué sur la variable.

4.5 Le Base Class Library (BCL)

Le .NET Framework contient un ensemble de classes contenues dans des DLL qui se nomment la BCL : Base Class Library, la BCL contient des milliers de types (classes) pour fournir différents services :

- gestion des collections comme des listes, des tables de hashage, des piles, des queues… ;
- des outils de diagnostic, en fournissant des services de traçage des événements, des compteurs de performances ou encore l'interaction avec d'autres processus ;
- des outils de globalisation et d'internationalisation ;
- la gestion des entrées/sorties ;
- la gestion de la sécurité ;
- la gestion des Web Services et des Web API ;

- la gestion des applications Web avec ASP.NET ;
- les applications Desktop avec WinForms et WPF ;
- les applications Console ;
- les services Windows ;
- la gestion du XML ou des flux I/O ;
- l'accès aux données avec ADO.NET.

Toutes les classes héritent de Object, qui est dans l'espace de nom (namespace) System. Pour avoir accès à une fonctionnalité du framework, il suffit d'incorporer l'assembly à son projet et d'inclure le namespace dans lequel est la fonctionnalité à l'aide du mot-clé using.

Par exemple :

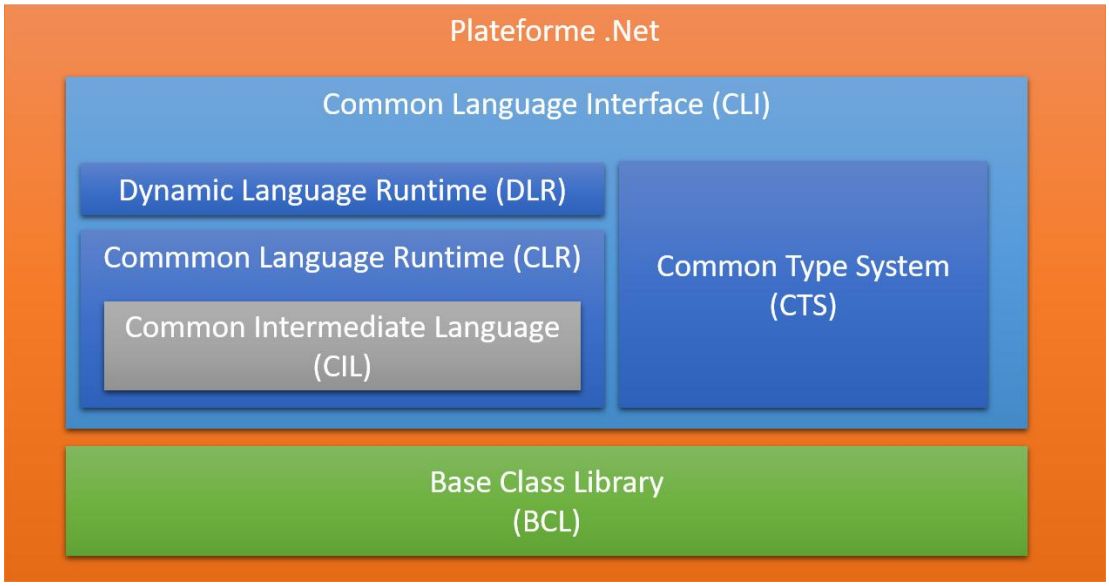
using System ;

using System.Xml ;

Voici quelques namespaces de la BCL :

Namespace	Description
System	Tous les types basic et services essentiels
Sytem.Data	Types d'accès aux données
Sytem.IO	Types sur les flux I/O, les fichiers et des répertoires
Sytem.Net	Types sur la communication bas-niveau
Sytem.Text	Types sur la gestion de l'ACSII, Unicode et l'encoding
Sytem.Threading	Types sur le multithreading
Sytem.Xml	Types sur la gestion du XML

Résumé :



La plateforme .Net regroupe :

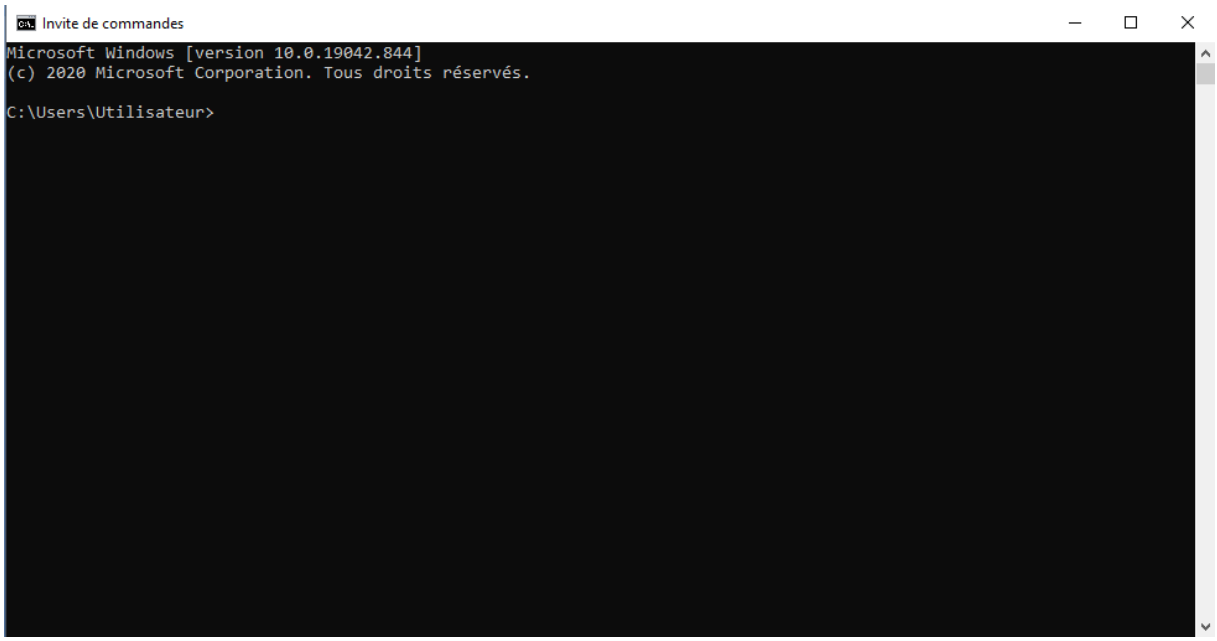
- une implémentation du Common Language Interface (CLI) ;

- la bilbiothèque standard (Base Class Library).

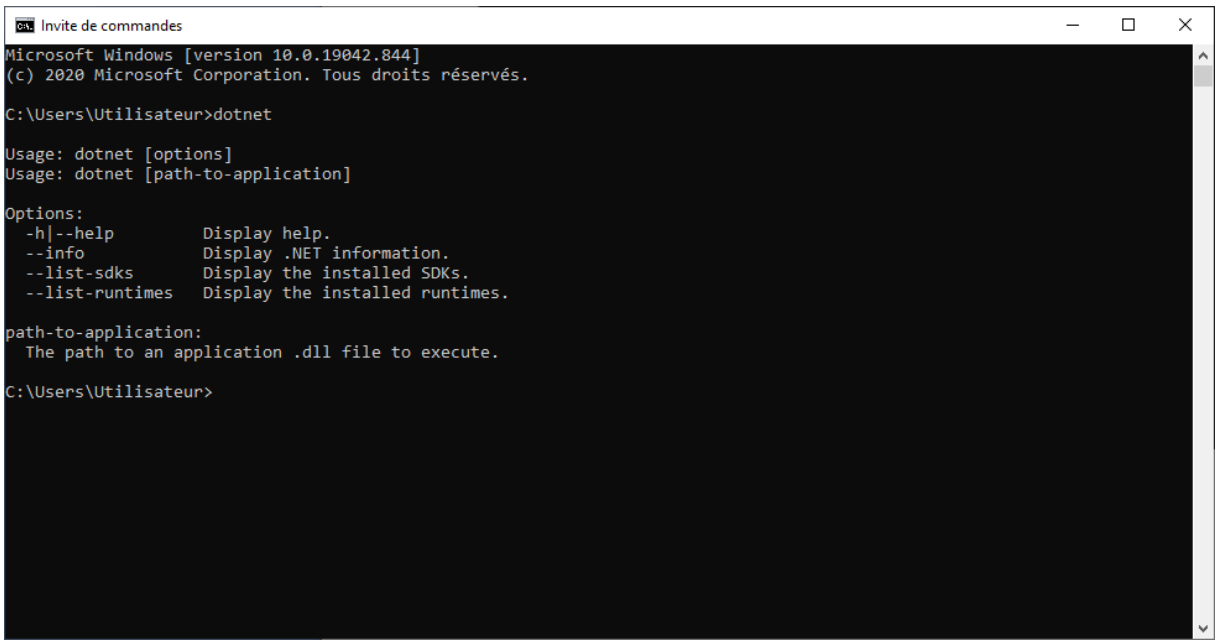
Annexe 1 : Comment savoir la version du framework installer sur mon Windows?

1ère étape : recherché la fenêtre “Invite commande”

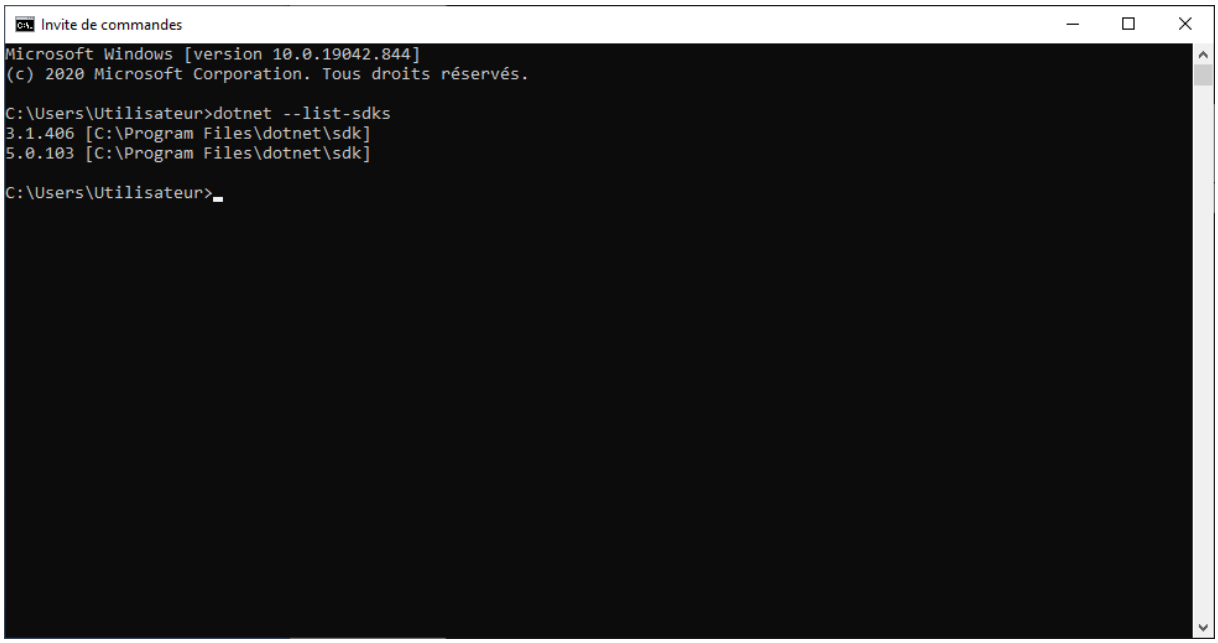
2ème étape :



Tapez “dotnet”



Puis “dotnet --list-sdks”



Nous pouvons voir ici que nous avons la version framework 3 NET et framework 5.0 NET.

Annexe 2 : Biographie

[1] Site de microsoft : https://docs.microsoft.com/fr-fr/dotnet/csharp/

[2] Aide mémoire C#, de Christophe Pichaud, janvier 2021, Edition DUNOD.