2019/2020

Cours de C#

[Sous-titre du document]

Noe Esparsa – Arthur Menu – Dimitri Dochy – Hector Courouble
ISEN LILLE

Table des matières

I. INTRODUCTION

Apparu du début des années 2000, le langage C# est un langage orienté objet créé par une équipe de programmeur dirigée par Anders Hejlsberg. Ce langage créé par Microsoft est issu d'une longue réflexion visant à compenser les défauts de la plupart des langages de programmation de l'époque (C++, Java, Delphi, Small Talk) liés au principe du Common Language Runtime (CLR).

Le CLR est un composant de la machine virtuelle du Framework .NET faisant tourner un bytecode précédant la transformation en code natif spécifique au système d'exploitation.

Depuis sa sortie en 2002, le langage a connu 7 versions différentes ajoutant de nouvelles possibilités au langage parmi lesquelles on peut citer :

- Les génériques sur la 2.0.
- LINQ sur la 3.0.
- Version compatible Linux sur la 6.0.

II. TYPES VALEURS & REFERENCES

Il existe deux genres de types en C# : les types référence et les types valeur.

Les variables des types référence font référence à leurs données (objets), tandis que les variables des types valeur contiennent directement leurs données.

Avec les types référence, deux variables peuvent faire référence au même objet. Ainsi, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable.

Avec les types valeur, chaque variable a sa propre copie des données et les opérations sur une variable ne peuvent pas affecter l'autre.

Il existe deux sortes de types valeur : Structures et énumérations. Ils sont tous deux implicitement dérivés de System.ValueType. Les types simples quant à eux sont un ensemble de types struct prédéfinis fournis par C#, composé des types suivants :

Types intégraux (entiers numérique et char), type virgule flottante et type Booleen.

Pour les types référence on utilise les mots clés suivants sont utilisés pour les déclarer :

- Classe / Interface / Delagate

Le langage C# fournit également les types référence intégrés suivants :

- Dynamic / Object / string

III. CHAINES DE CARACTERE & STRING BUILDER

Une chaîne de caractère est un objet de type String dont la valeur est du texte. Il n'y a pas de caractère de fin NULL à la fin de la chaîne, elle peut donc en contenir.

Initialisation d'un string :

```
String exemple = "chaine d'objet char";
```

On peut utiliser la classe StringBuilder pour créer une chaine de caractères et indiquer sa capacité maximum, si cette capacité vient à être dépassé, elle est automatiquement réallouée en la doublant.

Initialisation d'un string à l'aide d'un stringBuilder:

```
StringBuilder myString = new StringBuilder("Hello World!", 25);
```

Il est possible d'utiliser les méthodes append, insert, remove et replace. De plus un objet StringBuilder peut être convertit en String à l'aide de la méthode ToString.

IV. BOXING & UNBOXING

Le boxing est utilisé pour stocker des types valeur dans le tas rassemblé par garbage collection. Le boxing est une conversion implicite d'un type valeur en type object ou en un type interface implémenté par ce type valeur. Le boxing d'un type valeur alloue une instance d'objet sur le tas et copie la valeur dans le nouvel objet.

Dans l'exemple suivant, une variable de type valeur est déclarée :

```
int i = 123;
```

L'instruction ci-dessous réalise implicitement une opération de boxing sur la variable i :

```
// Boxing copies the value of i into object o.
object o = i;
```

```
int i = 123;
object o = (object)i; // explicit boxing
```

Le type valeur d'origine et que l'objet boxed utilisent des emplacements de mémoire distincts et peuvent, par conséquent, stocker des valeurs différentes.

L'unboxing est une conversion explicite du type object en un type valeur, ou d'un type interface en un type valeur qui implémente l'interface. Une opération d'unboxing comprend les étapes suivantes :

- Vérification de l'instance de l'objet pour s'assurer qu'il s'agit bien d'une valeur boxed du type valeur spécifié.
- Copie de la valeur de l'instance dans la variable de type valeur.

Les instructions suivantes expliquent les opérations de boxing et d'unboxing :

```
int i = 123; // a value type
object o = i; // boxing
 int j = (int)o; // unboxing
```

Pour que l'unboxing de types valeur réussisse au moment de l'exécution, l'élément qui est unboxé doit être une référence à un objet précédemment créé par boxing d'une instance de ce type valeur. La tentative d'extraction de null provoque un NullReferenceException. La tentative d'extraction d'une référence vers un type de valeur incompatible provoque un InvalidCastException.

V. LES CLASSES

Une classe définie un type d'objet mais il ne s'agit pas d'un objet.On peut les créer en utilisant le mot clé new suivi du nom de la classe que l'on veut créer.

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

Les classes prennent en charge l'héritage. Une classe peut hériter de toute autre interface ou classe.
Pour cela on crée une classe à partir d'une autre classe dont elle hérite des données et du comportement.

Ici la classe manager hérite de la classe employée :

```
public class Manager : Employee {    }
```

Une classe peut être déclarée abstract. Une classe abstraite contient des méthodes abstraites qui ont une définition de signature, mais aucune implémentation. Les classes abstraites ne peuvent pas être instanciées. Elles peuvent être utilisées uniquement à travers des classes dérivées qui implémentent les méthodes abstraites.

VI. LES INTERFACES (IEnumerable)

IEnumerable est l'interface de base pour toutes les collections non génériques qui peuvent être énumérées. Elle contient une méthode unique, GetEnumerator, qui retourne un IEnumerator qui offre la possibilité d'itérer au sein de la collection en exposant une propriété Current et des méthodes MoveNext et Reset.

Il est recommandé (bien que non obligatoire) d'implémenter IEnumerable et IEnumerator sur vos classes de collection pour activer la syntaxe foreach. Si votre collection n'implémente pas IEnumerable, vous devez toujours suivre le modèle d'itérateur pour prendre en charge cette syntaxe en fournissant une méthode GetEnumerator qui retourne une interface, une classe ou un struct.

VII. LES TYPES GENERIQUES

Les génériques sont introduits dans le .NET Framework 2.0, ils permettent de définir des structures de données de type sécurisé sans se limiter à un type de données réel.

Par exemple :

List<T> est une collection générique, elle peut être déclarée et utilisée avec n'importe quel type, List<int>, List<string>, etc…

Les génériques permettent un gain de performances car il n'y a pas de boxing et unboxing ce qui permet de gagner du temps.

De plus, avant l'implémentation des génériques, il n'y avait aucun moyen de connaitre le type de données contenu dans la liste. Ce problème est donc fixé, en effet une liste définie comme ceci : List<int> ne contiendra que des entiers.

```
//generic list

List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };

//non-generic list

ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
```

Un autre avantage de la connaissance du type d'un générique par le runtime est une meilleure expérience de débogage. Quand vous déboguez un générique en C#, le type de chaque élément dans la structure de données est connu. Sans les génériques, il ne serait pas possible de connaître le type de chaque élément.

VIII. LES DELEGUES

Un délégué est un type qui représente des références aux méthodes. Lorsque vous instanciez un délégué, vous pouvez associer son instance à toute méthode ayant une signature et un type de retour compatibles. Les délégués sont utilisés pour passer des méthodes comme arguments à d'autres méthodes. L'exemple suivant illustre une déclaration de délégué :

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

Toute méthode de n'importe quelle classe ou structure accessible qui correspond au type de délégué, peut être assignée au délégué.

Comparé à la surcharge des méthodes, les délégués incluent dans la signature la valeur de retour, il faut donc que la méthode à le même type de retour que le délégué.

L'un des intérêts majeurs des délégués est l'appelle à plusieurs méthode (ou multidiffusion). Ce concept repose sur la liste d'invocation, elle représente la liste des méthodes du delegate. Pour créer une liste d'invocation, il suffit de faire la somme des fonctions que l'on veut utiliser avec l'opérateur « + ». L'ajout d'une nouvelle méthode se fait par l'opérateur « += ».

Les délégués sont orientés objet et encapsulent une instance d'objet et une méthode. Ils permettent aux méthodes d'être transmises comme des paramètres.

Les méthodes anonymes et les expressions lambda sont également compilées en type de délégué.

Ici, un délégué privé est créé dans la classe *TestClass* et il permettra de pointer vers des méthodes qui ne retourne rien (*void*) et qui acceptent un tableau d'entier en paramètre.

```
class TestClass
{
    private delegate void myDelegate(int[] tableau);
}
```

Dans ce code, nous avons implémenté une méthode *private void TriAscendant(int[] tableau)*, qui a la même signature que notre délégué. Dans la méthode demoTri, nous commençons par déclarer une variable du type du délégué *myDeleguate*, qui est le délégué que nous avons créé. Puis nous faisons pointer cette variable vers la méthode *TriAscendant*.

```
class TestClass
{
    private delegate void myDelegate(int[] tableau);

    1 référence
    private void TriAscendant(int[] tableau)
    {
        Array.Sort(tableau);
    }

    0 références
    public void demoTri(int[] tableau)
    {
        myDelegate tri = TriAscendant;
        tri(tableau);
        foreach(int i in tableau)
        {
            Console.WriteLine(i);
        }
    }
```

On exécute la fonction Main() et on obtient bien le tableau trié.

```
class MainClass
{
    0 références
    static void Main()
    {
        int[] tableau = new int[] { 4, 1, 6, 10, 8, 5 };

        new TestClass().demoTri(tableau);
    }
}
```

IX. EXPRESSIONS LAMBDA

Tout d'abord une expression se caractérise par la succession de plusieurs opérandes/opérateurs ayant pour but de simplifier la déclaration de méthode, ou la surcharge d'opérateur de façon beaucoup plus concise. Les expressions lambda caractérisent une partie des expressions, celles-ci se déclarent de la façon suivante :

(input-parameters) => expression

```
() => SomeMethod()
```

```
(x, y) => x == y
```

```
(int x, string s) => s.Length > x
```

L'opérateur => est utilisé pour séparer les paramètres du corps de l'expression. Il s'agit de fonctions locales pouvant donc prendre 0 à plusieurs paramètres (dont le types peut être spécifié) leur but étant de réaliser dans la majeure partie des cas des opérations simples.

Les expressions lambdas sont capables de créer des délégués.

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

X. COLLECTIONS DE DONNEES

Les collections sont utilisées pour stocker et manipulés plus efficacement des données similaires. On peut utiliser la classe System.Array ou les classes présentes dans les espaces de noms System.Collection : System.Collections.Generic, System.Collections.Concurrent et System.Collections.Immutable pour ajouter, supprimer et modifier des éléments individuels ou une série d'éléments dans une collection.

Il existe les collections génériques et non-génériques. Les collections génériques fournissent des collections de type sécurisé au moment de la compilation. Pour cette raison, les collections génériques offrent généralement de meilleures performances. Les collections génériques acceptent un paramètre de type lorsqu'elles sont construites. Les collections non génériques stockent des éléments comme Object, nécessitent une conversion, et la plupart ne sont pas prises en charge pour le développement d'applications du Windows Store.

Toutes les collections fournissent des méthodes pour l'ajout, la suppression ou la recherche d'éléments au sein d'une collection.

Les collections du .NET Framework implémentent soit System.Collections.IEnumerable, soit System.Collections.Generic.IEnumerable<T> pour permettre à la collection d'être parcourue. Un énumérateur peut être vu comme un pointeur mobile pointant vers n'importe quel élément d'une collection.

Toutes les collections peuvent être copiées dans un tableau à l'aide de la méthode CopyTo. Toutefois, l'ordre des éléments du nouveau tableau sera basé sur l'ordre où ils sont retournés par l'énumérateur. Le tableau résultant est toujours unidimensionnel avec une limite inférieure de zéro.

La propriété Capacity d'une collection indique le nombre d'éléments qu'elle peut contenir. La propriété Count d'une collection indique le nombre d'éléments qu'elle contient réellement. Certaines collections masquent l'une de ces propriétés, voire les deux.

La capacité de la plupart des collections s'étend automatiquement quand la valeur de la propriété Capacity est atteinte. La mémoire est réallouée et les éléments sont copiés depuis l'ancienne collection vers la nouvelle.

XI. LES OPERATEURS

Les opérateurs en C# sont des fonctions spéciales fournissant les opérations primitives du langage. On en distingue 5 types : - Les opérateurs mathématiques - Les opérateurs booléens - Les opérateurs logiques - Les opérateurs d'attribution - Les opérateurs d'incrémentation

Opérateur C#	Signification
Calculs	
*	Multiplication
/	Division
%	Modulo
+	Addition

-	Soustraction
Tests	
is	Teste le type d'un objet
<	Inferieur à
>	Supérieur à
<=,>=	Inférieur ou égal, supérieur ou égal
!=	Différent de
==	Est égal à
Logique	
&&	ET logique
||	OU logique
??	Retourne l'opérande de gauche si non-null ou bien celui de droite
cond ? var1 : var2	Renvoie var1 si cond est vrai ou alors var2
Attribution	
=	donner une valeur à une variable
+=, -=,*=,/=,%=,&=,|=	contracter la variable de l'opérateur et de l'attribution
Incrémentation	
++	Ajoute 1 à la valeur
--	Enlève 1 à la valeur

XII. PROPRIETE & INDEXEURS

Une propriété est un membre qui fournit un mécanisme flexible pour la lecture, l'écriture ou le calcul de la valeur d'un champ privé. Les propriétés peuvent être utilisées comme s'il s'agissait de membres de données publics, mais ce sont en fait des méthodes spéciales appelées *accesseurs*. Elles permettent aux données d'être facilement accessibles tout en soutenant quand même la sécurité et la flexibilité des méthodes.

Les indexeurs permettent aux instances d'une classe ou d'un struct d'être indexés comme des tableaux. La valeur indexée peut être définie ou récupérée sans spécifier explicitement un membre de type ou d'instance. Les indexeurs s'apparentent aux propriétés à l'exception près que leurs accesseurs acceptent des paramètres.

Les indexeurs sont semblables aux propriétés. À l'exception des différences répertoriées dans le tableau suivant, toutes les règles définies pour les accesseurs des propriétés s'appliquent également aux accesseurs des indexeurs.

Propriété	Indexeurs
Permet aux méthodes d'être appelées comme si elles étaient des membres de données publics.	Permet aux éléments d'une collection interne d'un objet d'être accessibles à l'aide de la notation de tableau sur l'objet lui-même.
Accessible par le biais d'un nom simple.	Accessible par le biais d'un index.
Peut-être un membre statique ou un membre d'instance.	Doit être un membre d'instance.
Un accesseur get d'une propriété n'a aucun paramètre.	Un accesseur get d'un indexeur possède la même liste de paramètres formels que l'indexeur.
Un accesseur set d'une propriété contient le paramètre value implicite.	Un accesseur set d'un indexeur possède la même liste de paramètres formels que l'indexeur, outre le paramètre value.
Prend en charge la syntaxe abrégée avec les propriétés implémentées automatiquement.	Prend en charge les membres expression-bodied pour les indexeurs get-only.

XIII. LINQ

LINQ est l'acronyme de « Language-Integrated Query » permet d'effectuer des requêtes à des collections d'objet stockés en mémoire plus simplement. Pour effectuer une requête LINQ, il faut :

- Récupérer la source de données (une liste d'entier par exemple ou n'importe quelle collection d'objets)
- Créer une requête en utilisant les mots clés de l'API Standard Query Operator (Select, Where, …)
- Et enfin, exécuter la requête.

```
0 références
class IntroToLINQ
{
    0 références
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

Dans l'exemple précédent on voit que la requête est du type IEnumerable. Le type IEnumerable ainsi que toutes les interfaces qui dérivent de cette classe sont considérées comme des type requêtable. Une variable ayant un type requêtable est une source de données LINQ de base. Cependant il existe aussi des sources de données qui n'ont pas de type considéré comme requêtable, l'API qui gère LINQ de convertir ces types pour qu'ils deviennent requêtable.

Si les données sources ne sont pas déjà en mémoire en tant que type interrogeable, le fournisseur LINQ doit le représenter comme tel. Par exemple, LINQ to XML charge un document XML dans un type XElement Requêtable. LINQ permet aussi de créer des requêtes pour des bases de données SQL classique. Pour cela, il suffit de donner le type IQueryable (qui dérive de IEnumable) à la requête LINQ.

La requête

La requête spécifie les informations à récupérer à partir de la ou des sources de données. Si vous le souhaitez, une requête peut également spécifier la manière dont ces informations doivent être triées, regroupées et mises en forme avant d'être retournées. Une requête est stockée dans une variable de requête et initialisée avec une expression de requête. Pour faciliter l'écriture de requêtes, le langage C# propose désormais une nouvelle syntaxe de requête.

Comme pour le SQL, il y a plusieurs mots clés :

« from » : permet de spécifier l'endroit où se situe la source de donnée 23

« where » : permet de filtrer les éléments sélectionnés dans la source de donnée

« select » permet de préciser le type de retour des éléments

L'exécution

Une fois la requête construite, il faut pouvoir l'exécuter. Une requête LINQ possède deux types d'exécution :

- L'exécution différée

Dans le cas où l'exécution est différée, on peut exécuter la requête avec l'instruction « foreach » à n'importe quel moment dans un script C#. Coeme dans l'exemple ci-dessous :

```
// 3. Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

C'est dans l'instruction foreach que les résultats de requête sont récupérés. Par exemple, dans la requête précédente, la variable d'itération num contient chaque valeur (une à la fois) de la séquence retournée.

- L'exécution immédiate

On utilise une exécution immédiate lorsqu'on a besoin d'exécuter des fonctions d'agrégations sur le résultat de la requête. Les principales fonctions d'agrégations sont :

  - « Count » : permet de compter le nombre d'éléments retournés par la requête LNQ
  - « Max » : retourne l'élément maximum
  - « Average » : retourne la moyenne des éléments sélectionnés.
  - « First » : retourne le premier élément Remarque : Une exécution immédiate d'une requête n'utilise pas l'instruction « foreach ».

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

XIV. GESTION DES EXCEPTIONS

Les fonctionnalités de gestion des exceptions du langage C# aident à gérer les situations inattendues ou exceptionnelles qui se produisent lorsqu'un programme est en cours d'exécution. Elle utilise les mots clés *try*, *catch* et *finally* pour tenter des actions susceptibles de ne pas réussir, pour gérer les défaillances lorsque programme est susceptible de ne plus fonctionner et pour nettoyer ensuite les ressources. Les exceptions sont créées avec le mot clé *throw*.

Dans de nombreux cas, une exception peut être levée non pas par une méthode appelée directement par le code, mais par une autre méthode plus loin dans la pile des appels. Dans ce cas, le Common Language Runtime (CLR) déroule la pile à la recherche d'une méthode avec un bloc *catch* pour le type d'exception concerné et exécute le premier bloc *catch* de ce type qu'il trouve.

```
class ExceptionTest
{

    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }

    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Dans cet exemple, on cherche à savoir si le diviseur est nul afin de renvoyer l'exception dans le bloc catch. Ici b=0 donc la compilation du code va nous renvoyer l'exception : "Attempted divide by zero". Sans la gestion des exceptions, ce programme se terminerait avec une erreur DivideByZeroException non gérée.

Le mot clef « finally » est souvent utilisé pour libérer la mémoire ou pour fermer un FileStream par exemple plutôt que de laisser le Garbage collector s'en charger. Ce bloc sera exécuté dans tous les cas (qu'il y ait une erreur détectée ou non).

XV. ATTRIBUTS & REFLEXION

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 associée à une entité de programme, l'attribut peut être interrogé à l'exécution à l'aide d'une technique appelée réflexion.

Les attributs ajoutent des informations sur les types définis dans un programme. Vous pouvez ajouter des attributs personnalisés pour spécifier des informations supplémentaires si nécessaire.

```
[Serializable]
public class SampleClass {
        // Objects of this type can be serialized.
}
```

La réflexion est la capacité pour un programme à obtenir des informations de type ou des données représentant l'état de ce programme durant le runtime. Les classes permettant de les récupérer se situent dans l'espace de noms System.Reflection.

Elle est aussi utile pour examiner des types dans un assembly (bibliothèque de code compilé pour le déploiement, le versioning ou encore la sécurité). Tous les assembly programmés pour le CLR .NET comportent des metadata qui décrivent le contenu de l'assembly. .NET rend disponible plusieurs d'attributs standards, mais il est possible d'en créer d'autres. Outre le développeur, le CLR peut profiter de ces attributs à l'exécution même du programme. Les applications de la réflexion sont nombreuses, et nous celle-ci est dans la majeure partie des cas sans s'en rendre compte.

Exemple :

- On peut l'utiliser pour créer un explorateur d'assembly. Il permet de parcourir les modules, les types et leurs membres, et permet même de décompiler le CIL (Code Intermediate Language) en code C# ou .NET.

- On peut aussi l'utiliser pour la complétion de code. C'est grâce à la réflexion qu'il est possible d'utiliser cette fonctionnalité en .NET.