Cours : Programmation orientée objet en C#
Cote du cours : 26978
Session : Hiver 2022

Chapitre 08

Fondement de la Programmation OO – Partie 2

Table des matières

Introduction 2
I. Membres statiques 2
II. Classes et méthodes partielles 3
1. Classe Partielle 3
2. Méthode Partielle 4
3. Méthodes d'extension 5
III. Les espaces de noms 5
IV. Les indexeurs 6
Solution des exercices 8

Introduction

Dans cette seconde section, nous allons traiter plusieurs points. Nous commençons par les membres statiques, ensuite les champs et classes partielles. Après nous parlerons des indexeurs et on finira par les espaces de nom.

I. Membres statiques

Les membres statiques (appelé aussi membres partagés) d'une classe peuvent être des *variables*, des *propriétés* ou des *méthodes*. Ces membres ne sont pas spécifiques à un objet particulier d'une classe mais ils appartiennent à la classe elle-même. C'est-à-dire que tous les objets se partage le même membre statique et que sa modification par un des objets sera répercuter sur tous les objets. C'est pourquoi on l'atteint à travers le nom de la classe au lieu du nom de l'objet.

Vu que les membres statiques sont partagés avec tous les objets (existe en un seul exemplaire), ils ne peuvent pas accéder à un membre d'instance de sa classe. Pour définir un membre partagé en utilisant le mot-clé **static**.

Exemple : Nous voulons calculer le nombre de voiture qui ont été créées.

```
public class Voiture
{
   //Définition de la variable statique représentant le nombre de voiture.
    private static int nbVoiture = 0;
   //Définition de la propriété statique relative à la variable statique.
    public static int NbVoiture {
       get { return nbVoiture; }
       set { nbVoiture = value; }
    }
    ...
    public Voiture()
    {
         //Incrémentation du nombre de voitures à chaque création.
         NbVoiture++;
    }
}
```

Exemple : Nous voulons calculer le nombre de voiture qui ont été créées. Utilisation d'une propriété.

```
public class Voiture
{
   //Définition de la propriété statique relative à la variable statique.
    public static int NbVoiture { get; set; }
    ...
    public Voiture()
    {
         //Incrémentation du nombre de voitures à chaque création.
         NbVoiture++;
    }
```

```
}
```

Exercice 1 : Ajouter une variable statique pour calculer le nombre d'étudiants qui ont été créés.

Nous pouvons aussi définir des méthodes statiques. Ces méthodes sont conçues de façon à être appeler indépendamment de l'objet. Vous pouvez prendre à titre d'exemple les méthodes de la classe Math.

Exemple : Définir dans la classe Rationnel une méthode nommée somme permettant de calculer la somme de deux rationnels. Nous voulons que l'appel à la méthode somme soit indépendant de l'objet. La signature de la méthode est comme suit

public static Rationnel Somme (rationnel r1, Rationnel r2)

```
public static Rationnel Somme(Rationnel r1, Rationnel r2)
{
    return r1.Somme(r2);
}
```

Exercice 2 : Ajouter une méthode statique pour calculer le produit de deux rationnels et l'inverse d'un rationnel.

Le mot-clé **static** ne peut pas être combiné avec le mot-clé **const**.

II. Classes et méthodes partielles

1. Classe Partielle

Il est possible en C# de scinder une classe en plusieurs portions pouvant être réparties dans plusieurs fichiers : **Classe partielle**. Cette technique est généralement utilisée avec les interfaces graphiques. La déclaration d'une classe partielle se fait de la même façon qu'une classe classique, mais en préfixant le mot-clé **class** par le mot-clé **partial**. Par contre vous devez respecter les règles suivantes :

- Même nom de classe.
    - Même espace de nom.
    - À la compilation, les deux déclarations seront fusionnées.

Exemple : Classe adresse

```
namespace ChapitrePOO
{
    public partial class Adresse
    {
        private string code;
        private string ville;
    }
}
namespace ChapitrePOO
{
```

```
    public partial class Adresse
    {
        private string nomRue;
        private int numRue;
    }
}
```

2. Méthode Partielle

Le principe des **méthodes partielles** est relativement simple : une première partie de la méthode est définie avec le mot-clé partial mais elle ne possède pas de bloc de code associé. Ceci permet au compilateur de savoir que la méthode est susceptible d'avoir une implémentation dans une autre partie de la classe.
Pour écrire une méthode partielle vous devez respecter les règles suivantes :

- Les méthodes partielles doivent être définies dans des classes partielles sinon le compilateur génère une erreur.
- Les méthodes partielles ne peuvent pas être des fonctions.
- Les méthodes partielles ne peuvent pas avoir de modificateur d'accès (par défaut private).
- Une méthode partielle doit avoir la même signature que la première partie.
- Si la deuxième partie de cette méthode n'est pas présente, alors la méthode partielle n'a pas d'implémentation. Dans ce cas, le compilateur supprime la définition de la méthode partielle ainsi que les appels à cette méthode.

Exemple :

```
namespace ChapitrePOO
{
    public partial class Adresse
    {
        private string nomRue;
        private int numRue;

        partial void Afficher();
    }
}
namespace ChapitrePOO
{
    public partial class Adresse
    {
        private string code;
        private string ville;

        partial void Afficher() { Console.WriteLine("Adresse ..."); }

    }
}
```

3. Méthodes d'extension

Les méthodes d'extension permettent de rajouter des fonctionnalités à des classes ou des structures existantes **sans en modifier le code**.
Une méthode d'extension doit respecter une convention d'écriture stricte :

- Elle doit être définie dans une classe marquée static.
- Elle doit être marquée static.
- Son premier paramètre (obligatoire) doit être du type étendu. Il doit être précédé du mot-clé this.

Exemple : Ajouter une extension à la classe Rationnel pour définir le comportement de comparaison de deux Rationnels.

```
static class ExtensionsRationnel
{
    public static bool comparer(this Rationnel r, Rationnel r2)
    {
        return r.Den == r2.Den && r.Num == r2.Num;
    }
}

public static void Main()
{
    Rationnel r1 = new Rationnel(5, 4);
    Rationnel r2 = new Rationnel(5, 4);

    bool res = r1.comparer(r2);
    Console.WriteLine(res);

}
```

Exercice 3 : Ajouter une méthode à la classe string permettant de tester si un mot est un palindrome, c'est-à-dire s'il peut se lire indifféremment de gauche à droite ou de droite à gauche.

III. Les espaces de noms

Lorsque votre projet comporte plusieurs composants, il est possible de les organiser d'une façon logique à l'aide du concept de **namespace**. Un espace de nom est compose de plusieurs identificateurs séparé par des « . ».

Exemple :

```
System;
System.Collections.Generic;
System.Text;
```

Vous pouvez définir plusieurs classes avec le même nom dans des espaces de nom différents.

Pour inclure une classe dans un espace de noms, il suffit de déclarer le type à l'intérieur d'un **bloc namespace**.

```
namespace ChapitrePOO
{
    class Rationnel
    {
       //Code de la classe
    }
}
```

Pour utiliser une classe appartenant à un namespace particulier, il faut le déclarer dans le code à l'aide du mot-clé **using**.

Si on souhaite utiliser la classe List du namespace `System.Collections.Generic` alors il faut utiliser l'instruction suivante avant la déclaration du namespace :

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

Depuis C#6, le mot-clé using peut également être utilisé pour simplifier l'utilisation d'éléments statiques. Dans ce cas il faut combiner les mots-clés **using** et **static** comme présenter dans l'exemple suivant pour utiliser directement la méthode Pow :

```
using System;
using static System.Math;

namespace ChapitrePOO
{
    class StaticUsing
    {
        public static void Main()
        {
            double x = Pow(2, 5);
            Console.WriteLine(x); //2^5 = 32
        }

    }
}
```

IV. Les indexeurs

Les indexeurs sont un moyen simple pour accéder aux éléments d'un objet qui encapsule un tableau ou une collection. L'accès au contenu du tableau se fait par l'intermédiaire de l'index placer entre crochets avant le nom de l'objet. Pour définir un indexeur, il faut déclarer une propriété nommée **this** et spécifiant un ou plusieurs arguments entre crochets :

Dans l'exemple suivant nous allons déclarer une classe nommée Groupe. Un groupe est un ensemble d'étudiant (tableau). Nous allons utiliser un indexeur pour représenter le tableau et tester la classe dans une méthode main.

Exemple

```
namespace ChapitrePOO
{
    public class Groupe
    {
        //Déclaration et création du tableau
        Etudiant[] tab;

        public Groupe (int nbEtudiant)
        {
            tab = new Etudiant[nbEtudiant];
        }

        //Création de l'indexeur
        public Etudiant this[int index]
        {
            get { return tab[index]; }
            set { tab[index] = value; }
        }
    }
}


public static void Main()
{
    //Création d'un groupe de 10 étudiants
    Groupe g1 = new Groupe(10);

    //Affectation des 4 premiers étudiants
    g1[0] = new Etudiant("Bouhlel", "Mohamed", "Ottawa", 1);
    g1[1] = new Etudiant("Lalande", "Francois", "Gatineau", 1);
    g1[2] = new Etudiant("Lavoine", "Helene", "Montreal", 1);
    g1[3] = new Etudiant("Dion", "Celine", "Toronto", 1);

    //Affichage du premier étudiant
    g1[0].Afficher();
}
```

Exercice 4 :

- Créer une classe nommée Adresse. Une adresse est caractérisée par nomRue, numRue, ville, codePostal et provaince.
- Créer une classe nommée Personne. Une personne est caractérisée par un nom, prénom et des adresses. Utiliser un indexeur pour représenter le tableau d'adresses.

Solution des exercices

Exercice 1 :

```
public static int NbEtudiant { get; set; } = 0;


public Etudiant()
{
    NbEtudiant++;
    nom = "Bouhlel";
    prenom = "Mohamed";
}
```

Exercice 2 :

```
public static Rationnel Produit(Rationnel r1, Rationnel r2)
{
    return r1.Produit(r2);
}
public static Rationnel Inverse(Rationnel r1)
{
    return r1.Inverse();
}
```

Exercice 3 :

```
static class StringExtensions
{
    public static bool EstPalindrome(this string motATester)
    {
        if (motATester == null)
            return false;

        string motInverse = "";
        for (int i = motATester.Length - 1; i >= 0; i--)
        {
            motInverse = motInverse + motATester[i];
        }

        return motInverse.Equals(motATester);
    }
}
```

Exercice 4 :

```
public class Adresse
{
    private string nomRue, ville, codePostal, provaince;
    private int numRue;

    public Adresse(string nomRue, string ville, string codePostal,
string provaince, int numRue)
    {
        this.nomRue = nomRue;
        this.ville = ville;
        this.codePostal = codePostal;
        this.provaince = provaince;
        this.numRue = numRue;
    }

    public string NomRue { get { return nomRue; } set { nomRue =
value; } }
    public string Ville { get { return ville; } set { ville = value; }
}
    public string CodePostal { get { return codePostal; } set {
codePostal = value; } }
    public string Provaince { get { return provaince; } set {
provaince = value; } }
    public int NumRue { get { return numRue; } set { numRue = value; }
}

}

public class Personne
{
    private string nom, prenom;
    private Adresse[] adr;

    public string Nom { get { return nom; } set { nom = value; } }
    public string Prenom { get { return prenom; } set { prenom =
value; } }
    public Adresse this [int index]{ get { return adr[index]; } set {
adr[index] = value; } }



    public Personne(string nom, string prenom, int nb)
    {
        this.Nom = nom;
```

```
        this.Prenom = prenom;
        this.adr = new Adresse[nb];
    }
}
```