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

Chapitre 07

Fondement de la Programmation OO

Table des matières

Introduction

La programmation à base de fonction, comme on l'a réalisé jusqu'à maintenant, est appelée programmation procédurale. Elle est très répondue avec les langages tels que le langage C ou le langage Pascal. Le principe est de définir un programme comme un flot de données. Ce flot de données sera transformé, au fil de l'exécution, sous forme d'appel à des ***procédures*** et ***fonctions***. ce type de programmation ne présente aucun lien fort entre les données et les actions qui leurs sont associées.

La programmation orientée objet introduit la notion d'ensembles cohérents de données et d'actions : Objet. Cette notion d'objet est omniprésente lorsque l'on fait de la programmation orientée objet. Nous sommes en train d'assimiler tous les objets qui nous entoure à des objets ayant des propriétés et des actions qui leur sont associées. Ces objets peuvent aussi interagir (***communication***) ou être composés les uns des autres (**composition** et **agrégation**), ce qui permet de former des systèmes plus complexes.

Exemple :

- Un étudiant est caractérisé par son nom, son prénom, son adresse et son numéro d'inscription. Ces caractéristiques s'appellent des **propriétés**. Un étudiant peut faire les actions suivantes : poser des questions, répondre à des questions, discuter avec ces camarades, passer des évaluations… ces actions s'appellent des **comportements**.
- Un ordinateur est caractérisé par sa marque, son processeur, sa capacité stockage et sa mémoire. Un ordinateur peut faire les actions suivantes : installer un logiciel, désinstaller un logiciel, augmenter la mémoire, augmenter la capacité de stockage…
- Une voiture est caractérisée par sa marque, sa couleur et sa matricule. On peut démarrer la voiture, freiner, allumer les feux, tourner à droite, tourner à gauche, reculer, ...

L'existence de ces éléments (propriétés, comportement) dans le code C# se traduit par la notion de classes.

I. Notion de classe

Une classe est un **modèle** qui définit les ***propriétés*** et ***actions*** de chacun des objets qui seront créés à partir de lui. La majorité des types définis dans le framework .NET sont des classes. Dans cette section, nous allons voir en détail la création de la classe avec tous ces composants.

1- Déclaration :

Pour déclarer une classe on utilise le mot clé **class**. La syntaxe complète de déclaration d'une classe est comme suit :

```
modificateurAccès [ partial ] class NomClasse
[: ClasseBase, Interface1, Interface2, ...]
{
        /*    Code de la classe    */
}
```

Dans cette syntaxe il faut respecter les points suivants :

- Le nom d'une classe ne contient que des caractères alphanumériques ou le caractère _ et ne commence pas par un chiffre.

On recommande de nommer les classes en utilisant le style UpperCamelCase.

- La classe est délimitée par les caractères '{' et '}'.
- Les notions ClasseBase et interface feront l'objet respectivement des chapitres Héritage et Interface.
- Les modificateurs d'accès sont mots-clés définissant la visibilité de la classe pour le reste du code. Le tableau ci-dessous décrit les différents mots-clés applicables :

Mot-clé	Description
private	Vous ne pouvez l'utiliser que dans le cas de classe ***imbriquée*** (Chapitre classes imbriquées). La classe est visible uniquement par le type qui la contient.
protected	Vous ne pouvez l'utiliser que dans le cas de classe ***imbriquée*** (Chapitre classes imbriquées). la classe est visible par le type qui la contient et par ses types dérivés.
internal	la classe est visible par toutes les classes définies **dans** l'assembly. C'est le type par ***défaut***.
protected internal	la classe est visible par toutes les classes définies dans l'assembly et par les sous-classes extérieures à l'assembly (Chapitre Héritage).
public	la classe est visible par n'importe quel autre type, dans n'importe quel assembly.

Dans ce chapitre, nous allons étudier uniquement les modificateurs **internal** et **public**. Les autres modificateurs seront vus dans les chapitres suivants.

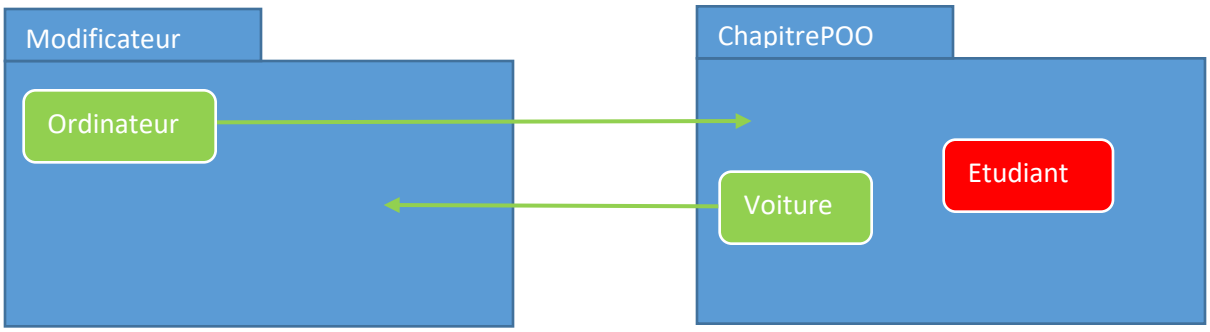
Exemple :

```
namespace ChapitrePOO
{
    public class Voiture
    {
    }
}
```

```
namespace ChapitrePOO
{
    internal class Etudiant
    {
    }
}
```

```
namespace Modificateur
{
    public class Ordinateur
    {
    }
}
```

Je n'ai pas le droit d'utiliser la classe **Etudiant** de l'assembly *ChapitrePOO* dans l'assembly *Modificateur* car elle est définie par le modificateur **internal**. Cependant, je peux utiliser la classe **Etudiant** dans toutes les classes de l'assembly *ChapitrePOO*.



2- Variables d'instance

Les classes possèdent des caractéristiques permettant d'identifier leurs propriétés. Ces propriétés sont appelées des ***variables membres***, ou ***membres d'instances***. La déclaration suit la syntaxe suivante :

```
modificateurs Type nomVariableMembre [ =valeurInitiale ] ;
```

Dans cette syntaxe il faut respecter les points suivants :

- Le nom d'une variable membre respecte les mêmes règles que le nom d'une variable.
- Une variable membre peut avoir une valeur initiale.
- Le modificateur peut être : modificateur d'accès (permettent de déterminer la ***visibilité*** des variables à l'extérieur de la classe.) ou une combinaison avec un autre modificateur tel que **const** ou **static** ou … la liste des modificateurs d'accès est présentée dans le tableau ci-dessous :

Mot-clé	Description
private	la variable est utilisable uniquement à l'**intérieur** de la classe dans laquelle elle est déclarée. Par défaut.
protected	la variable est utilisable à l'intérieur de la classe dans laquelle elle est déclarée ainsi que dans toutes ses classes filles (Chapitre Héritage).
internal	la variable est utilisable dans toutes les classes de l'assembly qui contient la déclaration de la variable.
protected internal	la variable est utilisable dans toutes les classes de l'assembly qui contient sa déclaration ainsi que dans toutes les classes filles de la classe à laquelle elle appartient (Chapitre Héritage).
public	la variable est utilisable sans restriction.
private protected	la variable est utilisable à l'intérieur de la classe dans laquelle elle est déclarée ainsi que dans toutes ses classes filles **déclarées au sein du même assembly** (Chapitre Héritage).

si aucun modificateur d'accès n'est spécifié alors **private** est affecté par défaut.

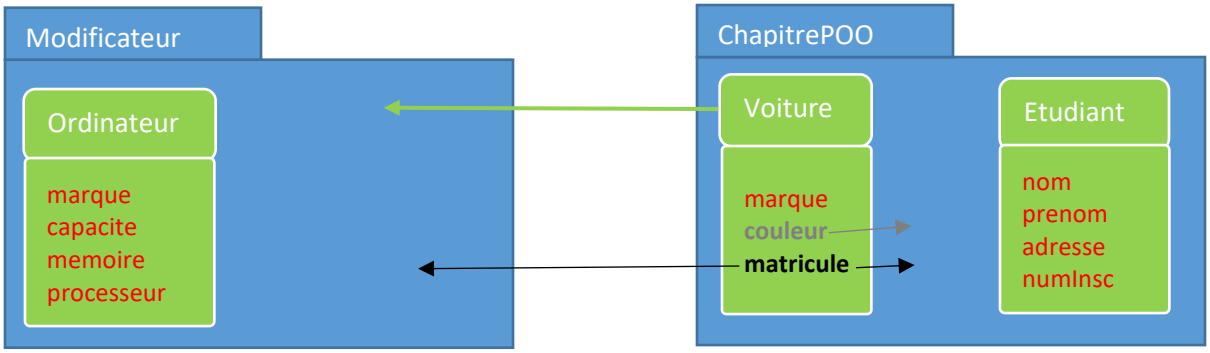
Exemple :

```
namespace ChapitrePOO
{
    public class Voiture
    {
        private string marque;
        internal string couleur;
        public string matricule;
    }
}
```

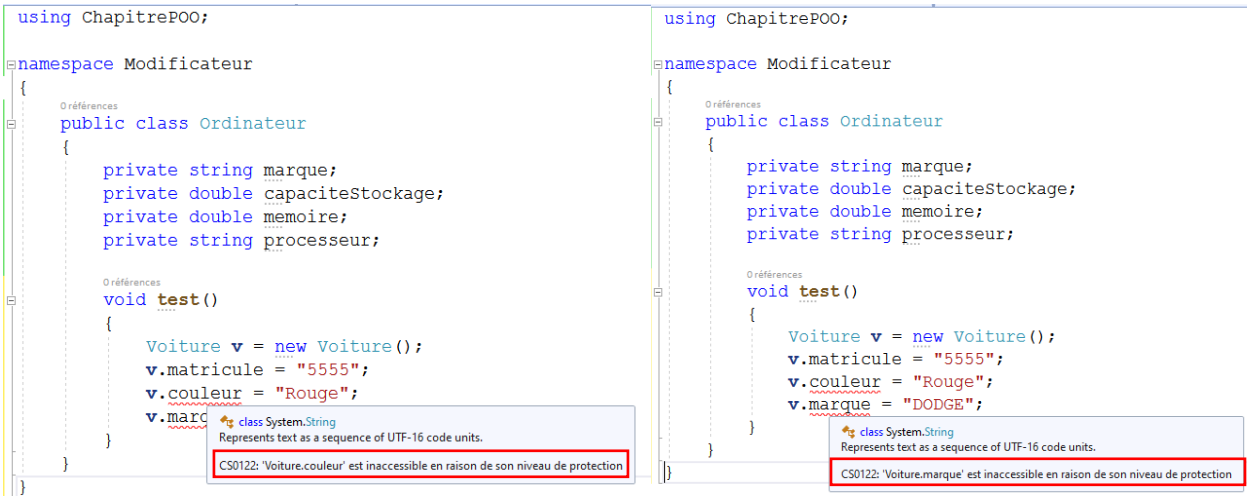
```
namespace ChapitrePOO
{
    public class Etudiant
    {
        private string nom;
        private string prenom;
        private string adresse;
        private int numInscription;
    }
}
```

```
namespace Modificateur
{
    public class Ordinateur
    {
        private string marque;
        private double capaciteStockage;
        private double memoire;
        private string processeur;
    }
}
```

Dans cet exemple, je ne peux pas utiliser les variables membre privées à l'extérieur de leurs classes respectives. Cependant, la variable membre couleur qui est déclarée avec le modificateur **internal** dans la classe Voiture de l'assembly ChapitrePOO est visible dans la classe Etudiant du même assembly et n'est pas visible dans la classe Ordianateur de l'assembly Modificateur. La variable membre Matricule de la classe voiture est visible dans toutes les classes de tous les assembly car elle est déclarée avec le modificateur **public**.



Exemple d'erreurs lors de l'exécution :



3- Les Constructeurs

Un constructeur est une méthode **spéciale** permettant l'instanciation d'un objet. Il sert généralement à initialiser les variables membres de la classe. Lors de la déclaration d'un constructeur il faut respecter les règles suivantes :

- Il porte le même nom que la classe
- Il ne possède aucun type de retour.
- Une classe possède au moins un constructeur.
- Si aucun constructeur n'est déclaré explicitement, le compilateur crée un constructeur vide n'acceptant aucun paramètre appelé constructeur par défaut.
- Si une classe fournit un constructeur, alors C# supprime le constructeur par défaut.
- Une classe peut avoir plusieurs constructeurs : surcharge de constructeur.

La **surcharge** de constructeur est le fait qu'une classe possède plusieurs constructeurs avec des arguments différents en types et / ou en nombres.

La syntaxe est comme suit :

```
modificateurAcces NomClasse ([parametre1, parametre2, …]) {
}
```

Exemple : Définition de 5 constructeurs pour la classe Etudiant

```
namespace ChapitrePOO
{
    public class Etudiant
    {
        private string nom;
        private string prenom;
        private string adresse;
        private int numInscription;
```

```
        public Etudiant() {
            nom = "Bouhlel";
            prenom = "Mohamed";
        }
        public Etudiant(string nomEtudiant)
        {
            nom = nomEtudiant;
        }
        public Etudiant(string nomEtudiant, string prenomEtudiant)
        {
            nom = nomEtudiant;
            prenom = prenomEtudiant;
        }
        public Etudiant(string nomEtudiant, string prenomEtudiant,
                string adresseEtudiant)
        {
            nom = nomEtudiant;
            prenom = prenomEtudiant;
            adresse = adresseEtudiant;
        }
        public Etudiant(string nomEtudiant, string prenomEtudiant,
                string adresseEtudiant, int numEtudiant)
        {
            nom = nomEtudiant;
            prenom = prenomEtudiant;
            adresse = adresseEtudiant;
            numInscription = numEtudiant;
        }
    }
}
```

Exemple : Définition de 3 constructeurs pour la classe Ordinateur

```
namespace Modificateur
{
    public class Ordinateur
    {
        private string marque;
        private double capaciteStockage;
        private double memoire;
        private string processeur;

        public Ordinateur()
        {
            capaciteStockage = 2048;
```

```
            memoire = 4096;
            marque = "HP";
        }

        public Ordinateur(string m, double cs, double memory)
        {
            capaciteStockage = cs;
            memoire = memory;
            marque = m;
        }
        public Ordinateur(string m, double cs, double memory,
                 string p)
        {
            capaciteStockage = cs;
            memoire = memory;
            marque = m;
            processeur = p;
        }

    }
}
```

Exercice 1 : Définir 4 constructeurs de la classe voiture.
Exercice 2 : Un rationnel est caractérisé par son numérateur et son dénominateur. Écrire la classe nommée rationnel en identifiant :

- Les attributs.
- 3 constructeurs.

4- Destructeur (ou finaliseur)

Lorsque le système a besoin de mémoire, le système de ramasse miette (Garbage collector) détruit les objets qu'il considère inutile en appelant leurs **destructeurs** s'ils en ont un. Un objet est jugé inutile lorsqu'il est orphelin (n'est plus référencé). Le rôle de cette méthode si elle existe est de libérer des ressources tel que la connexion à une base de données ou des fichiers.

Comme le constructeur, le destructeur est méthode particulière dont la signature est imposée.

- Son nom est le même que celui de la classe.
- Le nom est précédé du caractère ~.
- Il n'a pas de paramètre.
- Il n'a pas de type de retour.
- Pas de modificateur d'accès.

Exemple :

```
namespace ChapitrePOO
{
    public class Voiture
    {
        private string marque;
```

```
        internal string couleur;
        public string matricule;

        /*..........*/
        ~Voiture() {
            Console.writeLine("Fermetrure de l'objet");
    }
}
```

5- Création de méthodes

Une méthode est une fonction ou une procédure définie dans une classe. Elle définit un comportement/fonctionnalité associé à l'objet. Généralement les méthodes manipulent les variables membre de l'objet afin de modifier son état. La syntaxe de déclaration d'une méthode est comme suit :

```
modificateurs TypeRetour nomMethode ([argument1 , argument2, …])
{
        //corps de la méthode
        //Si le type de retour est différent de void, alors on rajoute une instruction
        //return à la fin.
}
```

Les modificateurs sont les mêmes que les variables membres.
Le type de retour peut être :

- void : la méthode ne retourne rien. Par exemple elle fait de l'affichage.
- Type primitif : tel que des entiers, des réels, …
- Type par référence : tableau ou les classes du framework ou les classes définies par l'utilisateur.

Si le type de retour de la méthode n'est pas **void**, alors on doit avoir une instruction « **return** » à l'intérieur.

Exemple : Ajouter à la classe Ordinateur les méthodes suivantes :

- Augmenter la mémoire de votre ordinateur par une valeur passée comme argument.
- Augmenter la capacité de stockage de votre ordinateur par une valeur passée comme argument.
- Afficher les propriétés d'un ordinateur.
- Conversion GO vers MO. La méthode retourne la capacité mémoire en MO.

```
namespace Modificateur
{
    public class Ordinateur
    {
        private string marque;
        private double capaciteStockage;
        private double memoire;
        private string processeur;
```

```
        public Ordinateur(){/*.....*/}
        public Ordinateur(string m, double cs, double memory){/*.....*/}
        public Ordinateur(string m, double cs, double memory, string p)
        {/*.....*/}

        public void AugmenterMemoire(double val)
        {
            memoire += val;
        }

        public void AugmenterDisque(double val)
        {
            capaciteStockage += val;
        }

        public void Afficher()
        {
            System.Console.WriteLine("Marque = {0}\n" +
                "Processeur = {1}\n" +
                "Memoire = {2}\n" +
                "Capacite de stockage = {3}.",
                marque, processeur, memoire, capaciteStockage);
        }

        public double Conversion_GO_MO()
        {
            return memoire * 1024;
        }
    }
}
```

Plusieurs méthodes peuvent avoir le **même** nom, mais elles doivent se distinguer par leurs arguments. C'est l'opération de surcharge. Il faut prendre en compte :

- Le nombre des arguments
- Le type des arguments
- L'ordre des arguments

Lors de la **surcharge** le nom des arguments et le type de retour ne font pas partie des critères d'une surcharge correcte.

Exemple :

- Surcharger la méthode augmentant la mémoire pour avoir en plus de la valeur son unité.
- Surcharger la méthode augmentant la capacité de stockage pour avoir en plus de la valeur son unité.

```
public void AugmenterMemoire(double val)
```

```
{
    memoire += val;
}
public void AugmenterMemoire(double val, string unite)
{
    double valGO;
    if (unite.Equals("TO"))
        valGO = val * 1024;
    else if (unite.Equals("MO"))
        valGO = val / 1024;
    else
        valGO = val;
    AugmenterMemoire(valGO);
}
public void AugmenterDisque(double val)
{
    capaciteStockage += val;
}
public void AugmenterDisque(double val, string unite)
{
    double valGO;
    if (unite.Equals("TO"))
        valGO = val * 1024;
    else if (unite.Equals("MO"))
        valGO = val / 1024;
    else
        valGO = val;
    AugmenterDisque(valGO);
}
```

6- Encapsulation

Chaque objet expose les données et les fonctions qui permettent aux autres objets d'interagir avec lui, et seulement celles-ci. Toutes les données propres au fonctionnement de l'objet et inutiles dans le cadre des interactions avec l'extérieur sont masquées. L'objet se comporte ainsi comme une **boîte noire** exposant seulement certaines données et fonctionnalités dont l'utilisation ne pourra pas corrompre l'état de l'objet.[1]

Pour mettre en pratique l'**encapsulation**, vous devez définir des méthodes d'accès aux variables membres privés. Ces méthodes sont les **get** et **set. Elles sont** appliquées à chaque variable membre pour former : **Propriété**.

[1] Livre : C#8 et visual studio 2019.

7- Les propriétés

Pour respecter le concept d'encapsulation, il est préférable d'exposer des **propriétés** plutôt que des variables membre. Les variables membres doivent être protéger de tous les accès non contrôlés et non autorisés. À cet effet, les propriétés ont été conçu. Elles permettent d'exécuter du code à chaque accès en **lecture** (**get**) ou en **écriture** (**set**) pour valider les données et maintenir l'objet dans un état cohérent.

Définition extraite de la documentation de C# [2]:

« Une propriété est un membre qui fournit un mécanisme flexible pour **lire**, **écrire** ou **calculer** la valeur d'un **champ privé**. Les propriétés peuvent être utilisées comme s'il s'agissait de membres de données publiques, mais ce sont en fait des méthodes spéciales appelées ***accesseurs*** . Cela permet d'accéder facilement aux données et contribue toujours à promouvoir la **sécurité** et la **flexibilité** des méthodes ».

La syntaxe de définition d'une propriété est comme suit :

```
private TypeMembre nomMembre;
public TypeMembre NomMembre {
        [ get {  /* Code du get */  } ]
        [ set {  /* Code du set */  } ]
}
```

- Un accesseur de propriété **get** est utilisé pour renvoyer la valeur de la propriété.
- Un accesseur de propriété **set** est utilisé pour attribuer une nouvelle valeur
- Ces accesseurs peuvent avoir différents niveaux d'accès.
- Le mot-clé « **value »** est utilisé dans le bloc **set**. Il représente la valeur assignée à la propriété.
- Une propriété automatique est une propriété simple qui ne nécessite aucun code d'accès personnalisé et peut être implémentée automatiquement par la syntaxe suivante :

```
public TypePropriete NomPropriete {
        get;set;
}
```

Exemple : définition de la propriété Mémoire de le la variable membre mémoire de la classe Ordinateur.

```
private double memoire;                 Variable membre privé.

public double Memoire                   Propriété publique.
{
    get { return memoire; }             Aucunes contraintes sur la lecture de la mémoire.
```

[2] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

```
    set { if (value > 0) memoire = value;
          else memoire = 128.0;
    }
}
```

On autorise uniquement des valeurs positives de la valeur.

L'écriture du **get** précédente est équivalente à celle de l'exemple ci-dessous car le bloc contient une seule instruction.

```
private double memoire;
public double Memoire
{
    get => memoire;

    set { if (value > 0) memoire = value;
        else memoire = 128.0;
    }
}
```

Écriture simplifiée de la méthode get. L'opération est possible car le bloc contient une seule instruction.

Une propriété peut être défini de plusieurs façon :

- *lecture-écriture* : elles ont à la fois un accesseur get et un set. Le cas de l'exemple précédent.
- *lecture seule* : elles ont un accesseur **get** mais pas d'accesseur **set**.
- *écriture seule* : elles ont un accesseur **set** mais pas d'accesseur **get**. Les propriétés en écriture seule sont rares et sont le plus souvent utilisées pour restreindre l'accès aux données sensibles.

Exemple : définition de la propriété Processeur en lecture seule de la variable membre processeur de la classe Ordinateur.

```
private string processeur;

public string Processeur
{
    get => processeur;
}
```

toute tentative d'assignation à la propriété a pour résultat une erreur de compilation

Exemple : définition de la propriété Solde en écriture seule de la variable membre solde de la classe Compte.

```
class Compte
{
    private double solde;

    public double Solde
    {
        set { solde = value; }
    }
}
```

```
Equivalent à:
public double Solde
{set => solde = value; }
```

Exemple : définition d'une propriété automatique. On va ajouter à la classe Ordinateur une propriété automatique nommée Proprietaire de type string indiquant le nom du propriétaire.

```
public string Proprietaire
{
    get; set;
}
```

C'est équivalent à :

```
private string proprietaire;
public string Proprietaire
{
    get => proprietaire;
    set => proprietaire = value;
}
```

Exercice 3 : Définir toutes les propriétés des variables membres de la classe Voiture. Respecter les contraintes suivantes :

- La marque est en majuscule.
- La couleur est une valeur parmi les suivantes : Rouge, Vert, Noir, Gris et Orange. La couleur par défaut est Rouge.

Exercice 4 : Définir toutes les propriétés des variables membres de la classe Etudiant. Respecter les contraintes suivantes :

- Le nom est en majuscule.
- Le prénom est en minuscule.
- Le numéro d'inscription est composé de 5 chiffres. La valeur par défaut est 11111.

Exercice 5 : Définir toutes les propriétés des variables membres de la classe Rationnel. Respecter les contraintes suivantes :

- Le dénominateur doit être différent de 0. La valeur par défaut est égale à 1.

Définir la méthode suivante :

- Afficher : permet d'afficher le rationnel sous la forme « num / den ».

II. Utilisation des classes

Une fois vos classes définies, la première chose qui nous vient à l'esprit est comment créer des objets à partir de nos classes. Un objet est une instance d'une classe. Une fois créé, il est possible de lire ou d'écrire des données dans l'objet, ainsi que d'exécuter les méthodes qui lui sont associées.

1- Instanciation

Pour créer des objets d'une classe, nous avons besoin du mot-clé new. Avec ce mot-clé, il faut utiliser un des constructeurs de la classe pour créer l'objet. Il faut vérifier les paramètres du constructeur lors de l'appel.

Exemple :

```
public static void Main()
{
    Voiture v1 = new Voiture();
    Voiture v2 = new Voiture("Dodge");
    Voiture v3 = new Voiture("BMW","Gris");
}
```

2- Utilisation de l'objet

C# utilise le symbole "." (point) pour permettre l'accès aux membres d'un objet. L'utilisation de l'objet suit la syntaxe suivante :

```
nomObjet . nomMethode(parametres);
nomObjet . nomPropriete
nomObjet . nomVariableMembre
```

Exemple :

```
public static void Main()
{
    Ordinateur ordi = new Ordinateur("HP", 2048, 1024, "Pentium 7");

    //Afficher l'ordinateur
    Console.WriteLine("Ordinateur 1 : ");
    ordi.Afficher();

    Ordinateur ordi1 = new Ordinateur();
    ordi1.Marque = "dell";
    ordi1.CapaciteStockage = 1024;
    ordi1.Memoire = 512;
    ordi1.Processeur = "CELERON";
    Console.WriteLine("\n");
    Console.WriteLine("Ordinateur 2 : ");
    ordi1.Afficher();
}
```

```
Ordinateur 1 :
Marque = HP
Processeur = Pentium 7
Memoire = 1024
Capacite de stockage = 2048.


Ordinateur 2 :
Marque = DELL
Processeur = CELERON
Memoire = 512
Capacite de stockage = 1024.
```

Vous pouvez créer vos objets en utilisant les initialiseurs. Ils permettent d'initialiser tout ou une partie de ses propriétés publiques. Le tout en une seule instruction.

Exemple :

```
Ordinateur ordi2 = new Ordinateur() {
      Marque = "HP", CapaciteStockage = 3072,
     Memoire = 1536, Processeur = "CELERON", Proprietaire = "Mohamed"
};
Console.WriteLine("\n");
Console.WriteLine("Ordinateur 3 : ");
ordi2.Afficher();
```

```
Ordinateur 3 :
Marque = HP
Processeur = CELERON
Memoire = 1536
Capacite de stockage = 3072.
```

III. L'objet courant

Le mot-clé "**this**" représente l'objet courant. Il renvoie l'instance de la classe dans laquelle il est utilisé. Il permet par exemple de passer une référence de l'objet courant à un autre objet. Les exemples ci-dessous montrent différentes façons d'utiliser le mot-clé this :

Exemple

```
public class Voiture
{
    private string marque;
    internal string couleur;
    public string matricule;

    public string Marque
    {
        get => this.marque;
        set => this.marque = value.ToUpper();
    }

    public string Couleur
    {
        get => marque;
        set {
            if (value.Equals("Rouge") || value.Equals("Vert") ||
value.Equals("Noir") || value.Equals("Gris") ||
value.Equals("Orange")) this.couleur = value;
            else this.couleur = "Rouge";
        }
    }
    public string Matricule {
        get => this.matricule;
        set => this.matricule = value;
    }
```

Utiliser le this pour faire appel à une variable membre.

```
    public Voiture(){}

    public Voiture(string marque)
    {
        this.Marque = marque;
    }

    public Voiture(string marque, string couleur)
    {
        this.Marque = marque;
        this.Couleur = couleur;
    }

    public Voiture(string marque, string couleur, string matricule)
    {
        this.Marque = marque;
        this.Couleur = couleur;
        this.Matricule = matricule;
    }

    ~Voiture() {
        //Mettre le code s'il existe...
    }

    public void Afficher()
    {
        Console.WriteLine(this.Matricule+"\n"+this.Couleur);
    }

    public void Afficher2()
    {
        Console.WriteLine(this.Marque + "\n");
        this.Afficher();
    }
}
```

Utiliser le this pour faire appel à une propriété.

Utiliser le this pour faire appel à une méthode.

On vous recommande d'utiliser le **this** autant que possible pour des questions de clarté.

Exercice 6 : Compléter la classe Rationnel pour définir les méthodes suivantes :

- **Somme** : Somme du rationnel courant avec le rationnel passé en paramètre. La méthode retourne un nouveau rationnel.
- **Somme** : surcharger la méthode Somme pour faire la somme entre le rationnel courant et le rationnel passé en paramètre sous la forme de deux entiers. La méthode retourne un nouveau rationnel.

- **Produit** : Produit du rationnel courant avec le rationnel passé en paramètre. La méthode retourne un nouveau rationnel.
- **Produit** : surcharger la méthode Produit pour faire le produit entre le rationnel courant et le rationnel passé en paramètre sous la forme de deux entiers. La méthode retourne un nouveau rationnel.
- **Inverse** : permet d'inverser le rationnel courant. La méthode retourne un nouveau rationnel.

Solution des exercices

Exercice 1 :

```
namespace ChapitrePOO
{
    public class Voiture
    {
        private string marque;
        internal string couleur;
        public string matricule;

        public Voiture()
        {
        }
        public Voiture(string m)
        {
            marque = m;
        }
        public Voiture(string m, string c)
        {
            marque = m;
            couleur = c;
        }
        public Voiture(string m, string c, string mat)
        {
            marque = m;
            couleur = c;
            matricule = mat;
        }
    }
}
```

Exercice 2 :

```
namespace ChapitrePOO
{
    class Rationnel
    {
        private int num; //Variable membre représentant le numérateur
        private int den; //Variable membre représentant le
dénominateur

        public Rationnel()
```

```
        {
            den = 1;
        }

        public Rationnel(int n)
        {
            num = n;
            den = 1;
        }
        public Rationnel(int n, int d)
        {
            num = n;
            den = d;
        }
    }
}
```

Exercice 3 :

```
public class Voiture
    {
        private string marque;
        internal string couleur;
        public string matricule;

        public string Marque
        {
            get => marque;
            set => marque = value.ToUpper();
        }

        public string Couleur
        {
            get => marque;
            set {
                if (value.Equals("Rouge") || value.Equals("Vert") ||
value.Equals("Noir") || value.Equals("Gris") ||
value.Equals("Orange")) couleur = value;
                else couleur = "Rouge";
            }
        }
        public string Matricule
        {
            get => matricule;
            set => matricule = value;
        }
……
```

```
}
```

Exercice 4 :

```
public class Etudiant
{
    private string nom;
    private string prenom;
    private string adresse;
    private int numInscription;

    public string Nom
    {
        get => nom;
        set => nom = value.ToUpper();
    }
    public string Prenom
    {
        get => prenom;
        set => prenom = value.ToLower();
    }
    public int NumInscription
    {
        get => numInscription;
        set
        {
            if (value >= 10000 && value <= 99999) numInscription =
value;
            else numInscription = 111111;
        }
    }
……
}
```

Exercice 5 :

```
class Rationnel
{
    private int num; //Variable membre représentant le numérateur
    private int den; //Variable membre représentant le dénominateur

    public int Num {
        get => num;
        set => num = value;
    }
```

```
    public int Den
    {
        get => den;
        set { if (value == 0) den = 1;
            else den = value;
        }
    }

……

}
```

Exercice 6 :

```
class Rationnel
{
    private int num; //Variable membre représentant le numérateur
    private int den; //Variable membre représentant le dénominateur

    public int Num {
        get => this.num;
        set => this.num = value;
    }

    public int Den
    {
        get => this.den;
        set { if (value == 0) this.den = 1;
            else this.den = value;
        }
    }

    public Rationnel()
    {
        Den = 1;
    }
    public Rationnel(int num)
    {
        this.Num = num;
        this.Den = 1;
    }
    public Rationnel(int num, int den)
    {
        this.Num = num;
        this.Den = den;
    }
```

```
    public Rationnel Somme(Rationnel r)
    {
        int numRes = r.Num * this.Den+ this.Num * r.Den;
        int denRes = r.Den * this.Den;
        Rationnel res = new Rationnel(numRes, denRes);
        return res;
    }

    public Rationnel Somme(int num, int den)
    {
        return this.Somme(new Rationnel(num, den));
    }
    public Rationnel Produit(Rationnel r)
    {
        int numRes = r.Num * this.Num;
        int denRes = r.Den * this.Den;
        Rationnel res = new Rationnel(numRes, denRes);
        return res;
    }

    public Rationnel Produit(int num, int den)
    {
        return this.Produit(new Rationnel(num, den));
    }

    public Rationnel Inverse()
    {
        return new Rationnel(this.Den, this.Num);
    }
}
```