Résumé Complet - C#

Programmation Orientée Objet

AmineGR03

28 janvier 2026

Table des matières

Introduction à la Programmation Orientée Objet

Concepts fondamentaux

Objet : Représente un concept, une idée ou toute entité du monde physique comme une voiture, une personne ou encore un livre, etc.

POO (Programmation Orientée Objet) : Un paradigme de programmation informatique qui consiste en la définition et l'assemblage de briques logicielles appelées objets.

Installation de Visual Studio

Pour développer en C#, il est nécessaire d'installer Visual Studio :

1. Télécharger Visual Studio Community depuis `https://visualstudio.microsoft.com/fr/downloads/`
2. Double-cliquer sur le programme d'installation
3. Dans Visual Studio Installer, cliquer sur "Modifier"
4. Sélectionner les charges de travail :
   - .NET Desktop development
   - ASP.NET and web development
5. Cliquer sur "Install"

Les Classes

Définition

Une **classe** est une manière de représenter un objet. En bref, c'est la structure d'un objet.

Caractéristiques importantes :

- En C#, la définition des attributs et leurs getters/setters est assez simplifiée
- La classe ne doit pas forcément avoir le même nom que le fichier
- Dans un fichier, on peut définir plusieurs classes
- C'est elle qui contient la méthode spéciale `Main()` qui sert de point d'entrée à l'application

Création d'une classe sous Visual Studio

1. Faire un clic droit sur le nom du projet dans l'Explorateur de solutions
2. Aller dans `Ajouter` → `Class`
3. Choisir `Class`
4. Saisir le nom de la classe (ex : `Voiture`) et valider

Exemple de structure de classe

```
namespace MaPremiereApplication
{
    class Voiture
    {
        // Attributs et m thodes ici
    }
}
```

Création d'Objets

Instanciation

Pour instancier une classe, il faut :

— Le nom de la classe

— Le nom de l'objet

— L'opérateur `new`

— Un constructeur de la classe

Exemple

```
static void Main(string[] args)
{
    Voiture voiture1 = new Voiture();
    Voiture voiture2 = new Voiture();
}
```

Ajout d'attributs à une classe

```
namespace MaPremiereApplication
{
    class Voiture
    {
        string Color;
        string Model;
        int Price;
    }
}
```

Remarque importante : Il est impossible d'affecter des valeurs aux attributs de l'objet `Voiture` car la visibilité par défaut, en C#, est `private`.

Notion de Visibilité

Modificateurs d'accès

Il existe plusieurs indicateurs de visibilité (peuvent être ajoutés avant attributs/classes/méthodes), mais les plus utilisés sont :

— `public` : Accessible partout

— `private` : Accessible uniquement dans la classe (défaut en C#)

— `protected` : Accessible dans la classe et ses dérivées

— `internal` : Accessible dans le même assembly

Exemple avec visibilité public

```
namespace MaPremiereApplication
{
    class Voiture
    {
```

```
        // D claration des attributs
        public string Color;
        public string Model;
        public int Price;
    }
}
```

Initialisation et affichage

```
static void Main(string[] args)
{
    Voiture voiture = new Voiture();
    voiture.Color = "red";
    voiture.Model = "Toyota";
    voiture.Price = 167000;

    // Affichage avec interpolation de cha nes
    Console.WriteLine($"Les caract ristiques sont: {voiture.Color} {
       voiture.Model}");
    Console.ReadKey();
}
```

La Méthode ToString()

Définition

La méthode `ToString()` permet d'afficher les détails d'un objet. Par défaut, elle retourne le nom complet de la classe.

Code généré par défaut

```
public override string ToString()
{
    return base.ToString();
}
```

Modification personnalisée

```
public override string ToString()
{
    return "Voiture [Color=" + Color + ", Model=" + Model +
           ", Price=" + Price + "]";
}
```

Accesseurs (Getters et Setters)

Propriétés avec accesseurs explicites

```
class Person
{
    private string name;
```

```
4
5       public string Name
6       {
7           get => name;
8           set => name = value;
9       }
10  }
```

Utilisation

```
1  Person person = new Person();
2  person.Name = "Sara";  // Le set accessor est invoqu  ici
3  Console.WriteLine($"Je m'appelle {person.Name}");
4  // Le get accessor est invoqu  ici
```

Propriétés auto-implémentées

Si les getters et setters ne contiennent pas un traitement particulier, on peut utiliser des propriétés auto-implémentées :

```
1  public class Person
2  {
3      public string Name { get; set; }
4  }
```

Propriétés en lecture seule / écriture seule

— **Lecture seule** : Supprimer le `set`

```
1      public int Age { get; }
```

— **Écriture seule** : Supprimer le `get`

```
1      public int Age { set; }
```

Exemple avec validation

```
1  public class Personne
2  {
3      private int age;
4
5      public int Age
6      {
7          get { return age; }
8          set
9          {
10             if (value > 0 && value < 100)
11                 age = value;
12             else
13                 throw new ArgumentException("L' ge  doit  tre  entre 0
                       et 100");
14         }
15     }
16 }
```

Les Constructeurs

Définition

Un **constructeur** est une méthode particulière portant le nom de la classe et ne retournant aucune valeur.

Caractéristiques :

- Toute classe en C# a un constructeur par défaut sans paramètre
- Ce constructeur sans paramètre n'a aucun code
- On peut le définir si un traitement est nécessaire
- La déclaration d'un objet de la classe fait appel à ce constructeur sans paramètre
- On peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs

Exemple de constructeur avec paramètres

```
public class CompteBancaire
{
    private string titulaire, devise;
    private double solde;

    // Constructeur
    public CompteBancaire(string Titulaire, double Solde, string Devise)
    {
        this.titulaire = Titulaire;
        this.solde = Solde;
        this.devise = Devise;
    }
}
```

Utilisation

```
static void Main(string[] args)
{
    CompteBancaire compte = new CompteBancaire("Jad", 1700000, "Euro");
    Console.WriteLine(compte);

    CompteBancaire compte2 = new CompteBancaire("Rhizlane", 200000, "MAD
        ");
    Console.WriteLine(compte2);
}
```

Important : En définissant un constructeur avec paramètres, le constructeur par défaut (sans paramètre) n'existe plus automatiquement.

Solution : Définir les deux constructeurs

```
public class CompteBancaire
{
    public string titulaire, devise;
    public double solde;

    // Constructeur avec param tres
```

```
    public CompteBancaire(string Titulaire, double Solde, string Devise)
    {
        this.titulaire = Titulaire;
        this.solde = Solde;
        this.devise = Devise;
    }

    // Constructeur sans param tres
    public CompteBancaire() { }
}
```

Attributs et Méthodes Statiques

Concept

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs. Si nous désirions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe Personne), on utilise un **attribut statique** ou **attribut de classe**.

Définition d'un attribut statique

```
public class Personne
{
    public static int NbrPersonnes { get; set; }

    // Autres attributs...
}
```

Incrémentation dans les constructeurs

```
public Personne(int num, string name)
{
    this.num = num;
    this.name = name;
    NbrPersonnes++;  // Incr menter le compteur
}

public Personne()
{
    NbrPersonnes++;  // Incr menter le compteur
}
```

Utilisation

```
static void Main(string[] args)
{
    Console.WriteLine(Personne.NbrPersonnes);  // Affiche 0

    Personne personne1 = new Personne(200, "Ahmed Ahmed");
    Console.WriteLine(Personne.NbrPersonnes);  // Affiche 1

    Console.WriteLine(personne1);
```

```
}
```

Remarque : Les attributs et méthodes statiques sont accessibles directement via le nom de la classe, sans avoir besoin d'instancier un objet.

L'Héritage

Définition

L'**héritage** est utilisé lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes). Il consiste à créer une nouvelle classe dite **classe dérivée** ou **classe fille** à partir d'une classe existante dite **classe de base** ou **classe parente** ou **classe mère**.

Avantages de l'héritage

L'héritage permet de :

— Récupérer le comportement standard d'une classe objet (classe parente) à partir de propriétés et de méthodes définies dans celle-ci

— Ajouter des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée

— Modifier le comportement standard d'une classe d'objet (classe parente), en surchargeant certaines méthodes de la classe parente dans la classe dérivée

Exemple conceptuel

— Un `Enseignant` a un ID, un nom, un prénom et un salaire

— Un `Etudiant` a aussi un ID, un nom, un prénom et une note

— Sémantiquement, `Enseignant` et `Etudiant` sont une sorte de `Personne`

— Les deux partagent plusieurs attributs tels que ID, nom et prénom

— On peut mettre en commun les attributs ID, nom et prénom dans une classe `Personne`

— Les classes `Etudiant` et `Enseignant` hériteront de la classe `Personne`

Syntaxe de l'héritage

Classe parente

```
class ClassA
{
    // Propri t  de la classe
    public int DataA;

    // M thode de la classe
    public int FonctionA1()
    {
        // Code de la fonction fonctionA1
    }

    // M thode red finissable
    public virtual int FonctionA2()
    {
        // Code de la m thode fonctionA2
    }
}
```

Classe fille

```
class ClassB : ClassA
{
    // Propri t  de la classe
    public int DataB;

    // Red finition de la m thode de la classe parente
    public override int FonctionA2()
    {
        // Code de la fonction fonctionA2
    }

    // Nouvelle m thode
    public int FonctionB1()
    {
        // Code de la m thode fonctionB1
    }
}
```

Exemple complet : Personne, Enseignant, Etudiant

Classe Personne

```
namespace MonProjet
{
    class Personne
    {
        public int Num { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }
    }
}
```

Classe Enseignant

```
namespace MonProjet
{
    class Enseignant : Personne
    {
        public int Salaire { get; set; }
    }
}
```

Classe Etudiant

```
namespace MonProjet
{
    class Etudiant : Personne
    {
        public string Niveau { get; set; }
    }
}
```

Redéfinition de ToString()

Dans la classe Enseignant

```
public override string ToString()
{
    return base.ToString() + " Enseignant [salaire=" + Salaire + "]";
}
```

Dans la classe Etudiant

```
public override string ToString()
{
    return base.ToString() + " Etudiant [niveau=" + Niveau + "]";
}
```

Remarque : Le mot-clé `base` permet d'appeler une méthode de la classe mère.

Constructeurs avec héritage

Classe Enseignant avec constructeur

```
class Enseignant : Personne
{
    public Enseignant(int num, string nom, string prenom, int salaire)
        : base(num, nom, prenom)
    {
        Salaire = salaire;
    }

    public Enseignant() { }

    public int Salaire { get; set; }

    public override string ToString()
    {
        return base.ToString() + " Enseignant [salaire=" + Salaire +
            "]";
    }
}
```

Classe Etudiant avec constructeur

```
class Etudiant : Personne
{
    public Etudiant(int num, string nom, string prenom, string niveau)
        : base(num, nom, prenom)
    {
        Niveau = niveau;
    }

    public Etudiant() { }

    public string Niveau { get; set; }

```

```
    public override string ToString()
    {
        return base.ToString() + " Etudiant [niveau=" + Niveau + "]";
    }
}
```

Polymorphisme

Un objet de la classe `Personne` peut être créé à partir d'une classe dérivée :

```
// Correct
Enseignant enseignant2 = new Enseignant(4, "Adib", "Abdellah", 40000);

// Correct aussi (polymorphisme)
Personne enseignant2 = new Enseignant(4, "Adib", "Abdellah", 40000);

// Incorrect -    ne pas faire !
Enseignant enseignant2 = new Personne(4, "Adib", "Abdellah", 40000);
```

Opérateur is

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `is` :

```
Console.WriteLine(enseignant2 is Enseignant);  // Affiche True
Console.WriteLine(enseignant2 is Personne);   // Affiche True
Console.WriteLine(personne is Enseignant);    // Affiche False
```

Concepts Clés à Retenir

Modificateurs de visibilité

- `public` : Accessible partout
- `private` : Accessible uniquement dans la classe (défaut)
- `protected` : Accessible dans la classe et ses dérivées
- `internal` : Accessible dans le même assembly

Mots-clés importants

- `new` : Instanciation d'un objet
- `override` : Redéfinition d'une méthode virtuelle
- `virtual` : Méthode redéfinissable dans une classe dérivée
- `base` : Accès aux membres de la classe parente
- `static` : Membres de classe (partagés par toutes les instances)
- `is` : Vérification du type d'un objet

Propriétés vs Attributs

- **Attributs** : Variables de classe (peuvent être privées)
- **Propriétés** : Encapsulation avec get/set (recommandé)
- **Propriétés auto-implémentées** : { `get; set;` }

Exercices Pratiques

Exercice 1 : Classe Personne

Définir une classe `Personne` avec trois propriétés (Nom, Prénom et Age). L'Age doit être supérieur à 0 et inférieur à 100.

Exercice 2 : Classe Cercle

Créer un constructeur `Cercle` qui crée un cercle avec un rayon fourni par un argument. Les cercles construits doivent avoir deux méthodes `GetAire()` et `GetPerimetere()` qui donnent à la fois l'aire et le périmètre respectifs.

Formules :

— Aire $= \pi \times r^2$

— Périmètre $= 2 \times \pi \times r$

Conclusion

La Programmation Orientée Objet en C# permet de structurer le code de manière modulaire et réutilisable. Les concepts fondamentaux (classes, objets, héritage, encapsulation) sont essentiels pour développer des applications maintenables et extensibles.