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

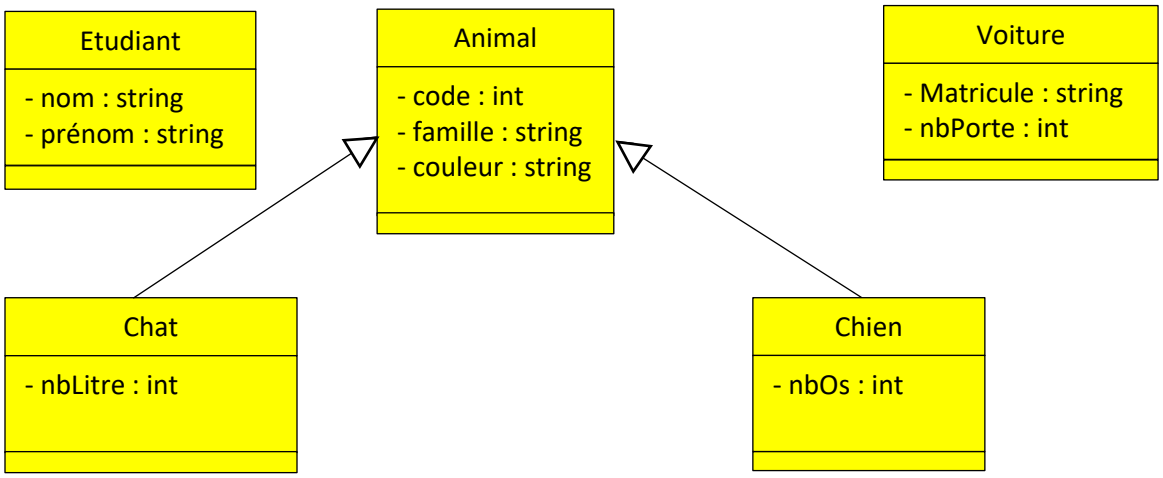
# Chapitre 12

# Les interfaces

## Table des matières

## Introduction

Pour illustrer la notion d'interface, nous allons considérer l'exemple ci-dessous :




On sait que tous les animaux font des sons différents. Mais, les voitures, les bateaux et les personnes font aussi des sons différents des animaux. Comment faire pour :

- Implémenter cette fonctionnalité dans les classes héritant de la classe animal, la classe voiture, la classe bateau et la classe personne?
- Garantir que toutes les classes implémentent le même comportement?
- Savoir que la classe a implémenté la fonctionnalité?
- Dire aux autres classes tel que lecteur, téléphone d'implémenter ce comportement?

La réponse à toutes ces questions est l'utilisation des **interfaces**.

# I. Création d'interface

## 1. Définition

Une interface est un contrat définissant les **données** et les **comportements** communs entre différents types d'objets. En C# on appelle **interface** ce genre de contrat.

Concrètement, une interface est un **type** qui possède **uniquement** des membres **publics**. De façon générale, ces membres ne possèdent pas d'implémentation (abstract). C'est aux types qui signent le contrat qui doivent fournir une implémentation.

Les interfaces représentent un des piliers de la programmation orientée objet car ça représente une couche d'abstraction particulièrement utile pour rendre une application **modulaire**.

## 2. Syntaxe

La syntaxe de déclaration d'une interface est comme suit :

```
<Modificateur Accès> interface INomInterface [: Interfaces de base]
{
        /*les membres de l'interfaces*/
}
```

La déclaration de l'interface suit les règles suivantes :

- Utiliser le mot clé interface et non class.
- Uniquement des membres publics. Aucun modificateur de visibilité n'est autorisé sur ces membres.
- Généralement, les membres n'ont pas d'implémentation.

Par convention, le nom des interfaces en C# commence par un **I**.

### 3. Exemple

L'interface de l'exemple introductif s'écrit :

```
public interface ISon
{
    void Parler();
}
```

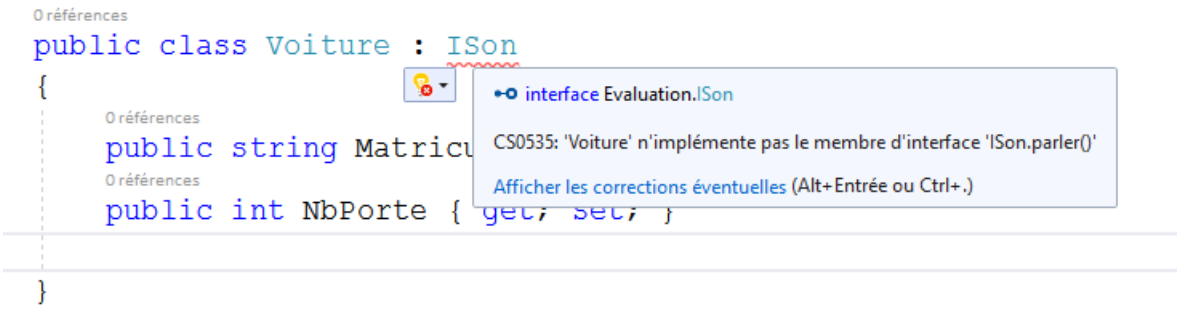
## II. Utilisation d'interface

Une fois l'interface définie, c'est au tour des classes de dire qu'elles vont utiliser (signer le contrat) l'interface. Pour cela, on utilise la même syntaxe que pour l'héritage :

```
<Modificateur Accès> class NomClasse [: Interface1, Interface2…]
{
        /*… */
}
```

On dit alors que la classe **implémente** l'interface. Lorsqu'une classe implémente une interface elle doit implémenter tous les membres de l'interface sinon vous aurez une erreur de compilation.

**Exemple :**



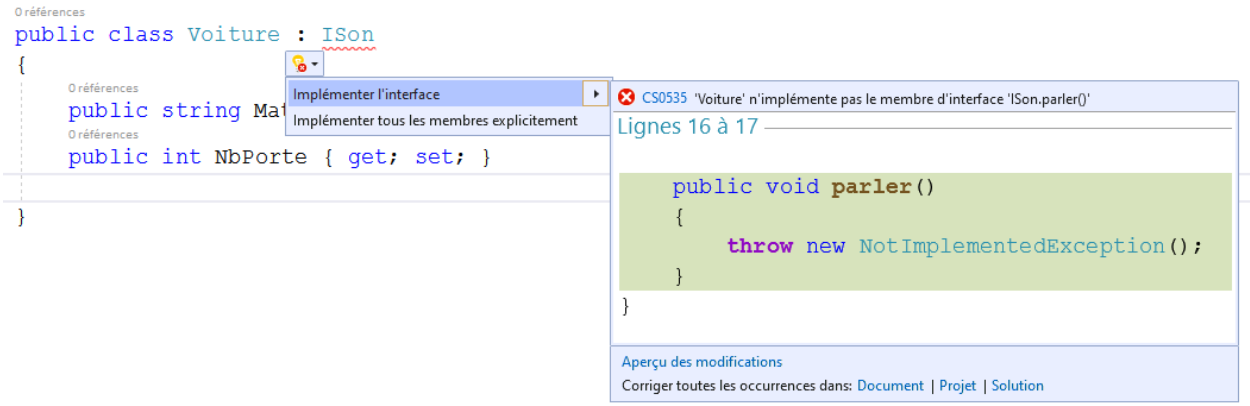

Pour résoudre ce problème on implémente les méthodes de l'interface ISon. Deux façons d'implémentation sont possibles : **Implicite** et **explicite**

### 1. Implémentation implicite

Dans la grande majorité des cas, les interfaces sont implémentées de manière implicite. Ceci signifie simplement que chacun des membres définis dans l'interface existe dans les classes implémentant l'interface.

- Visibilité publique
- Pour les méthodes même signature que celle définie dans l'interface.

Vous pouvez générer le code directement en vous servant de l'IDE.



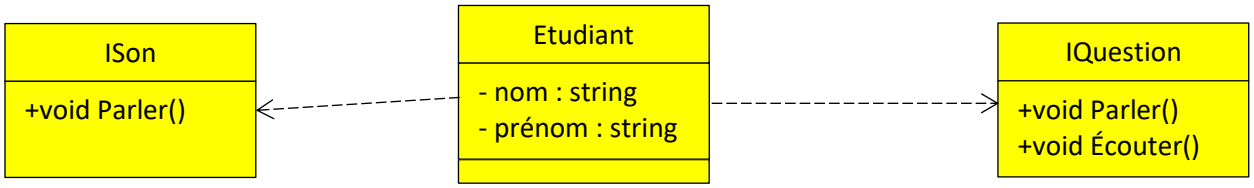

Une fois le code généré, vous pouvez supprimer l'instruction « `throw new NotImplementedException();` » et la remplacer par votre propre implémentation de la méthode.

```
public class Voiture : ISon
{
    public string Matricule { get; set; }
    public int NbPorte { get; set; }
    public void Parler()
    {
        Console.WriteLine("La voiture fait VROMVROMVROOOOOOM");
    }
}
```

## 2. Implémentation explicite

Dans des cas particuliers, il se peut que votre classe implémente le même comportement provenant de deux interfaces différentes. Dance cas, si vous utilisez l'implémentation implicite ça revient à partager la même implémentation avec toutes les interfaces concernées! Si on veut donner des implémentations différentes à chaque méthode de chaque interface on procède dans ce cas à une implémentation explicite.

**Exemple :**




Dans cet exemple, la classe Etudiant implémente les deux interfaces ISon et IQuestion. Les deux interfaces contiennent une méthode Parler. Nous voulons donner une implémentation différente dépendamment de l'interface.

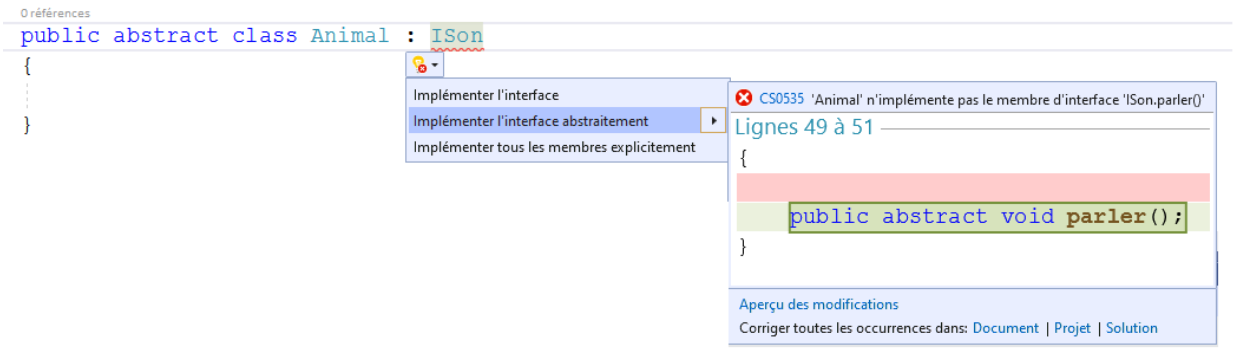
```
public interface ISon
{
    void parler();
}
public interface IQuestion
{
    void parler();
    void Ecouter();
}
public class Etudiant : ISon, IQuestion
{
    public string Nom { get; set; };
    public string Prenom { get; set; };

    void IQuestion.Ecouter()
    {
        Console.WriteLine("J'écoute la réponse .....");
    }

    void ISon.parler()
    {
        Console.WriteLine("Je parle ....");
    }

    void IQuestion.parler()
    {
        Console.WriteLine("Je pose ma question ...");
```

```
    }
}
```

### 3. Implémenter l'interface abstraitement

Une troisième façon d'implémenter l'interface est de le faire de façon abstraite. Dans ce cas la classe implémentant l'interface est une classe abstraite et les méthodes de l'interfaces sont aussi abstraites.

Exemple : Pour la classe Animal, je sais que tous les animaux produisent des sons, mais je ne peux connaitre le son que si je connais l'animal en question. D'où la classe Animal est une classe abstraite. Il ne suffit pas de déclarer la classe abstraite, mais il faut aussi écrire la méthode de l'interface et lui ajouter le mot clé abstract à sa signature. Sinon une erreur de compilation se produit :




Une fois on connait l'animal en question on peut implémenter la méthode parler.

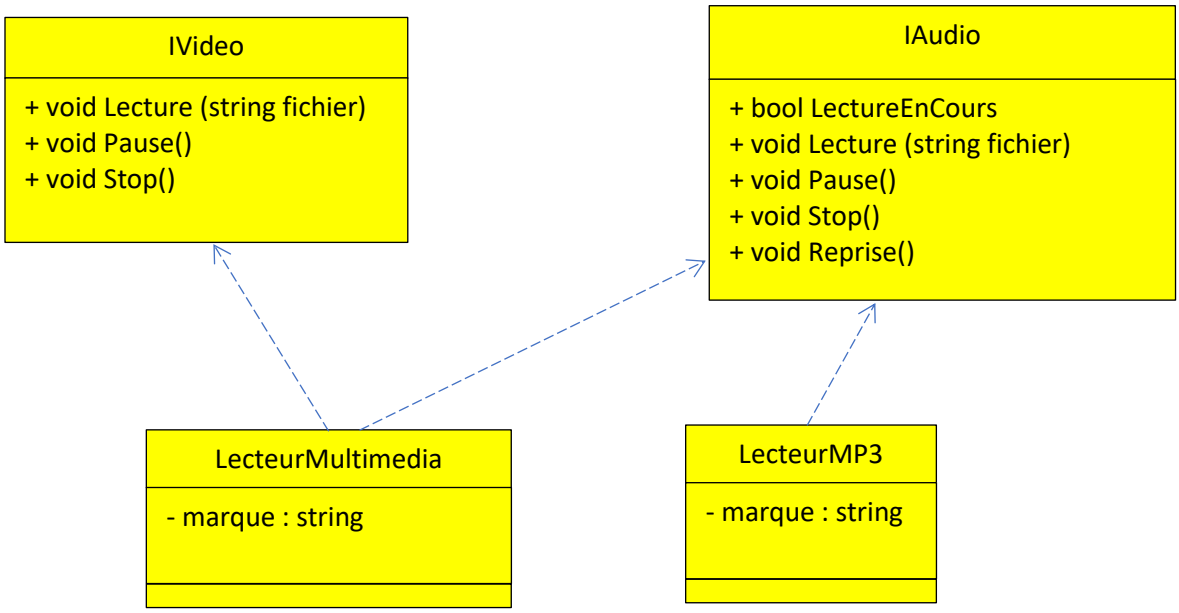
```
public abstract class Animal : ISon
{
    public abstract void parler();
}

public class Chien : Animal
{
    public int NbOs { get; set; }
    public override void parler()
    {
        Console.WriteLine("Je produit le son WOOF WOOF");
    }
}
public class Chat : Animal
{
    public int NbLitre { get; set; }
    public override void parler()
    {
        Console.WriteLine("Je produit le son MIOO MIOO");
```

```
    }
}
```

## III. Exercice

Soit le schéma suivant, essayer de fournir une implémentation des différentes classes du diagramme de classe.

La LectureEnCour est une propriété accessible en lecture.




Solution :

```
public interface IAudio
{
    bool LectureEnCours { get; }
    void Lecture(string fichier);
    void Pause();
    void Stop();
    void Reprise();
}
public interface IVideo
{
    void Lecture(string fichier);
    void Pause();
    void Stop();
}
public class LecteurMultimedia : IAudio, IVideo
```

```
{
    private bool enCours;
    bool IAudio.LectureEnCours { get { return enCours; }  }

    void IAudio.Lecture(string fichier)
    {
        Console.WriteLine("Lecture Audio ....");
    }

    void IVideo.Lecture(string fichier)
    {
        Console.WriteLine("Lecture Vidéo ....");
    }

    void IAudio.Pause()
    {
        Console.WriteLine("Lecture Audio en pause ....");
    }

    void IVideo.Pause()
    {
        Console.WriteLine("Lecture Vidéo en pause ....");
    }

    public void Reprise()
    {
        Console.WriteLine("Reprise de la Lecture Audio ....");
    }

    void IAudio.Stop()
    {
        Console.WriteLine("Arrêt de la Lecture Audio ....");
    }

    void IVideo.Stop()
    {
        Console.WriteLine("Arrêt de la Lecture Vidéo ....");
    }
}

public class LecteurMP3 : IAudio
{
    private bool mp3EnCours;
    bool IAudio.LectureEnCours { get { return mp3EnCours; }  }

    public void Lecture(string fichier)
```

```
    {
        Console.WriteLine("Lecture Audio ....");
    }

    public void Pause()
    {
        Console.WriteLine("Lecture Audio en pause ....");
    }

    public void Reprise()
    {
        Console.WriteLine("Reprise de la Lecture Audio ....");
    }

    public void Stop()
    {
        Console.WriteLine("Arrêt de la Lecture Audio ....");
    }
}
```

## IV. Méthodes concrètes dans les interfaces

Depuis C# 8[1], les interfaces ont la possibilité de porter des implémentations par défaut pour leurs membres. L'objectif avoué de cette fonctionnalité est de faciliter l'évolution de librairies de code en simplifiant la gestion de la compatibilité descendante. La forme la plus simple de cette fonctionnalité est la possibilité de déclarer une *méthode concrète* dans une interface, qui est une méthode avec un corps.

**Exemple :**

```
public interface IAudio
{
    bool LectureEnCours { get; }
    void Lecture(string fichier);
    void Pause();
    void Stop();
    void Reprise();
    void Lecture30Seconde() {
        Console.WriteLine("Je joue une piste de 30 secondes...");
    }
}
```

Vous n'êtes pas obligé d'implémenter cette méthode dans les classes qui implémente l'interface.

[1] https://docs.microsoft.com/fr-ca/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods

## V. Polymorphisme

Le polymorphisme est souvent considéré comme le troisième pilier d'une programmation orientée objet, après l'encapsulation et l'héritage. Le polymorphisme est le mot grec qui signifie « plusieurs formes » et il prend deux aspects distincts :

- Au moment de l'exécution, les objets d'une classe dérivée peuvent être traités comme des objets d'une classe de base dans les paramètres de méthode et les collections ou les tableaux. Lorsque ce polymorphisme se produit, le type déclaré de l'objet n'est plus identique à son type au moment de l'exécution.

**Exemple :**

```
public static void Main(String[] args)
{
    CompteCourant cc11 =
     new CompteCourant("Hamdi", 25000, "CAD", "123456", 100);
    CompteCourant cc12 =
     new CompteCourant("George", 16000, "CAD", "523698", 1000);
    CompteEpargne cc21 =
     new CompteEpargne("Mohamed", 10000, "CAD", 10);
    CompteEpargne cc22 =
     new CompteEpargne("Monica", 63200, "CAD", 5);

    List<CompteBancaire> banque = new List<CompteBancaire>();
    banque.Add(cc11);
    banque.Add(cc21);
    banque.Add(cc12);
    banque.Add(cc22);

}
```

- Les classes de base peuvent définir et implémenter des *méthodes* virtuelles, et les classes dérivées peuvent les substituer, ce qui signifie qu'elles fournissent leur propre définition et implémentation. Au moment de l'exécution, quand le code client appelle la méthode, le compilateur recherche le type au moment de l'exécution et appelle cette substitution de la méthode virtuelle. Dans votre code source, vous pouvez appeler une méthode sur une classe de base et provoquer l'exécution de la version d'une classe dérivée de la méthode.

**Exemple :**

```
public static void Main(String[] args)
{
    CompteCourant cc11 =
     new CompteCourant("Hamdi", 25000, "CAD", "123456", 100);
    CompteCourant cc12 =
     new CompteCourant("George", 16000, "CAD", "523698", 1000);
    CompteEpargne cc21 =
     new CompteEpargne("Mohamed", 10000, "CAD", 10);
```

```
    CompteEpargne cc22 =
     new CompteEpargne("Monica", 63200, "CAD", 5);

    List<CompteBancaire> banque = new List<CompteBancaire>();
    banque.Add(cc11);
    banque.Add(cc21);
    banque.Add(cc12);
    banque.Add(cc22);

    foreach (CompteBancaire cb in banque)
    {
        Console.WriteLine(cb);
    }
}
```

```
Type de compte: CompteCourant
Titulaire : Hamdi
Solde : 25000 CAD

Type de compte: CompteEpargne
Titulaire : Mohamed
Solde : 10000 CAD

Type de compte: CompteCourant
Titulaire : George
Solde : 16000 CAD

Type de compte: CompteEpargne
Titulaire : Monica
Solde : 63200 CAD
```

**Exemple 2 :**

On veut créer une application de dessin qui permet à un utilisateur de créer différents types de formes sur une surface de dessin. Vous ne savez pas au moment de la compilation les types de formes spécifiques que l'utilisateur va créer. Cependant, l'application doit conserver une trace des différents types de formes créés et les mettre à jour en réponse aux actions de la souris. Vous pouvez utiliser le polymorphisme pour résoudre ce problème en deux étapes :

1. Créer une hiérarchie de classe dans laquelle chaque classe de forme dérive d'une classe de base commune.
2. Utiliser une méthode virtuelle pour appeler la méthode appropriée dans une classe dérivée via un seul appel à la méthode de classe de base.

```
public class Forme
{
    public virtual void Dessiner() {
        Console.WriteLine("Je me prépare à dessiner");
    }
```

```
}

public class Rectangle : Forme
{
    public override void Dessiner()
    {
        base.Dessiner();
        Console.WriteLine("Je dessine un Rectangle\n");
    }
}

public class Cercle : Forme
{
    public override void Dessiner()
    {
        base.Dessiner();
        Console.WriteLine("Je dessine un Cercle\n");
    }
}

public class Triangle : Forme
{
    public override void Dessiner()
    {
        base.Dessiner();
        Console.WriteLine("Je dessine un Triangle\n");
    }
}

public class Carre : Forme
{
    public override void Dessiner()
    {
        base.Dessiner();
        Console.WriteLine("Je dessine un Carre\n");
    }
}
public class TestForme {
    public static void Main(String[] args)
    {
        List<Forme> formes = new List<Forme>();
        formes.Add(new Carre());
        formes.Add(new Cercle());
        formes.Add(new Rectangle());
        formes.Add(new Triangle());
        formes.Add(new Cercle());
```

```
        formes.Add(new Rectangle());

        foreach (Forme f in formes)
            f.Dessiner();
    }
}
```