C# : POO

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

C#

Particularité de **C#**

- La redéfinition et la surcharge sont possibles.
- La définition des attributs (appelés champs) et leurs getters/setters (appelés propriétés) est assez simplifiée.
- La classe ne doit pas forcément avoir le même nom que le fichier.
- Dans un fichier, on peut définir plusieurs classes.
- Le mot-clé `this` n'est pas obligatoire en l'absence d'ambiguïté.

C#

(Quatre) niveaux de visibilité :

- `private` : accessible seulement à l'intérieur de la classe (**par défaut**)
- `public` : accessible par tout (à l'intérieur et à l'extérieur de la classe)
- `protected` : accessible seulement à l'intérieur de la classe ou à partir d'une classe dérivée.
- `internal` : accessible uniquement à l'assembly actuel.

C#

Commençons par créer un nouveau projet dans `MaSolution`

- Dans l'`Explorateur de solutions`, faire clic droit sur `MaSolution`
- Aller à `Ajouter > Nouveau projet`
- Sélectionner `Application console`
- Cliquer sur `Suivant`
- Remplir les champs
  - `Nom` avec `CoursPoo`
  - `Solution` avec `MaSolution`
- Valider

Déclaration

```
visibility class ClassName
{
   ...
}
```

© Achref EL MOUELHI ©

Déclaration

```
visibility class ClassName
{
   ...
}
```

Les membres d'une classe

- Champs (attributs)
- Constructeur(s) (et destructeur)
- Méthodes
- Propriétés : méthodes accessibles comme si elles étaient des champs (exemple : getters et setters)
- Indexeurs
- Opérateurs
- ...

C#

Pour créer une classe sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom du projet dans l'`Explorateur de solutions`
- Aller dans `Ajouter > Class`
- Choisir `Classe`
- Saisir `Personne` dans `Nom :` et valider

C#

Contenu de `Personne.cs`

```
namespace CoursPoo
{
    class Personne
    {
    }
}
```

C#

Ajoutons les trois attributs suivants à la classe `Personne`

```
namespace CoursPoo
{
    class Personne
    {
        int num;
        string nom;
        string prenom;
    }
}
```

C#

Pour créer des objets à partir d'une classe (instancier la classe)

- Le nom de la classe
- Le nom de l'objet
- L'opérateur `new`
- Un constructeur de la classe

© Achref EL M

C#

Pour créer des objets à partir d'une classe (instancier la classe)

- Le nom de la classe
- Le nom de l'objet
- L'opérateur `new`
- Un constructeur de la classe

Dans la méthode `Main` de `Program.cs`, on instancie la classe `Personne`

```
Personne personne = new Personne();
```

© Achref EL Mo

C#

Pour créer des objets à partir d'une classe (instancier la classe)

- Le nom de la classe
- Le nom de l'objet
- L'opérateur `new`
- Un constructeur de la classe

Dans la méthode `Main` de `Program.cs`, on instancie la classe `Personne`

```
Personne personne = new Personne();
```

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

C#

Pour accéder aux attributs, on peut leur attribuer la visibilité
`public`

```
namespace CoursPoo
{
    class Personne
    {
        public int num;
        public string nom;
        public string prenom;
    }
}
```

C#

Dans `Main()`, on peut affecter des valeurs aux attributs d'un objet de la classe `Personne`

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne();
            personne.num = 100;
            personne.nom = "wick";
            personne.prenom = "john";
            Console.WriteLine($"Je m'appelle { personne.prenom } { personne.nom }");
            Console.ReadKey();
        }
    }
}
```

C#

Depuis C#9, l'appel du constructeur peut être simplifié

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new();
            personne.num = 100;
            personne.nom = "wick";
            personne.prenom = "john";
            Console.WriteLine($"Je m'appelle { personne.prenom } { personne.nom }");
            Console.ReadKey();
        }
    }
}
```

C#

Création d'objet à partir d'une classe : deuxième méthode

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                nom = "wick",
                prenom = "john",
                num = 100
            };
            Console.WriteLine($"Je m'appelle { personne.prenom } { personne.nom }");
            Console.ReadKey();
        }
    }
}
```

C#

Et si on essaye d'afficher l'objet `personne`

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                nom = "wick",
                prenom = "john",
                num = 100
            };
            Console.WriteLine(personne);
            Console.ReadKey();
        }
    }
}
```

C#

Et si on essaye d'afficher l'objet `personne`

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                nom = "wick",
                prenom = "john",
                num = 100
            };
            Console.WriteLine(personne);
            Console.ReadKey();
        }
    }
}
```

En exécutant, le résultat est

```
CoursPoo.Personne
```

C#

Pour afficher les détails d'un objet, il faut que la méthode `ToString()` soit implémentée

Pour générer `ToString()` sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Générer les substitutions`
- Sélectionner `ToString()` et valider

© Achref

C#

Pour afficher les détails d'un objet, il faut que la méthode `ToString()` soit implémentée

Pour générer `ToString()` sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Générer les substitutions`
- Sélectionner `ToString()` et valider

© Achref

Code généré de la méthode `ToString()`

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

Modifions le contenu de `ToString()`

```
public override string ToString()
{
    return "Personne [num=" + num + ", nom=" + nom + ", prenom=" + prenom + "]";
}
```

© Achref EL MOUEL

C#

Modifions le contenu de `ToString()`

```
public override string ToString()
{
    return "Personne [num=" + num + ", nom=" + nom + ", prenom=" + prenom + "]";
}
```

bref EL MOU

En exécutant, le résultat est

```
Personne [num=100, nom=wick, prenom=john]
```

C#

Quelques conventions en programmation orientée objet

- Attributs non-publiques
- Getter et setter publiques pour chaque attribut

C#

Mettons la visibilité `private` à tous les attributs de la classe `Personne`

```
namespace CoursPoo
{
    class Personne
    {
        private int num;
        private string nom;
        private string prenom;
    }
}
```

C#

Mettons la visibilité `private` à tous les attributs de la classe `Personne`

```
namespace CoursPoo
{
    class Personne
    {
        private int num;
        private string nom;
        private string prenom;
    }
}
```

Compilation ⇒ erreur

Attributs privés ⇒ inaccessibles dans `Main`.

C#

Pour générer les getters/setters sous **Visual Studio Community 2022**

- Sélectionner le nom d'attribut dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Encapsuler le champ`

Ou bien

- Sélectionner le nom d'attribut dans l'éditeur de code
- Aller dans `Edition > Refactoriser > Encapsuler le champ (et utiliser la propriété)`
- Valider en cliquant sur `Appliquer`

C#

Générons les getters et setters dans `Personne`

```
namespace CoursPoo
{
    class Personne
    {
        private int num;
        private string nom;
        private string prenom;

        public int Num { get => num; set => num = value; }
        public string Nom { get => nom; set => nom = value; }
        public string Prenom { get => prenom; set => prenom = value; }

        public override string ToString()
        {
            return "Personne [num=" + Num + ", nom=" + nom + ", prenom=" + prenom + "]"
               ;
        }
    }
}
```

C#

Générons les getters et setters dans `Personne`

```
namespace CoursPoo
{
    class Personne
    {
        private int num;
        private string nom;
        private string prenom;

        public int Num { get => num; set => num = value; }
        public string Nom { get => nom; set => nom = value; }
        public string Prenom { get => prenom; set => prenom = value; }

        public override string ToString()
        {
            return "Personne [num=" + Num + ", nom=" + nom + ", prenom=" + prenom + "]"
               ;
        }
    }
}
```

`value` : mot-clé permettant de récupérer la valeur affectée à la propriété.

C#

Pour utiliser les getters et setters dans `Main()`

```
Personne personne = new Personne()
{
    Nom = "wick",
    Prenom = "john",
    Num = 100
};
Console.Write($"Je m'appelle { personne.Prenom } { personne.Nom }");
```

C#

Si les getters et setters ne contiennent pas un traitement particulier, on peut simplifier les getters et setters précédents

```
namespace CoursPoo
{
    class Personne
    {
        private int num;
        private string nom;
        private string prenom;

        public int Num { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }

        public override string ToString()
        {
            return "Personne [num=" + num + ", nom=" + nom + ", prenom=
               " + prenom + "]";
        }
    }
}
```

C#

Rien ne change pour l'appel

```
Personne personne = new Personne()
{
    Nom = "wick",
    Prenom = "john",
    Num = 100
};
Console.Write($"Je m'appelle { personne.Prenom } { personne.Nom }");
```

C#

Nous pouvons même supprimer la déclaration des attributs et utiliser directement les propriétés (nous mettons à jour le `ToString()`)

```
namespace CoursPoo
{
    class Personne
    {

        public int Num { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }

        public override string ToString()
        {
            return "Personne [num=" + Num + ", nom=" + Nom + ", prenom=
               " + Prenom + "]";
        }
    }
}
```

C#

Rien ne change pour l'appel

```
Personne personne = new Personne()
{
    Nom = "wick",
    Prenom = "john",
    Num = 100
};
Console.Write($"Je m'appelle { personne.Prenom } { personne.Nom }");
```

C#

En supprimant le `set` : le numéro devient accessible seulement en lecture

```
public class Personne
{
    public int Num { get; }
}
```

C#

Pour initialiser la propriété avec une valeur par défaut [C# 6.0]

```
public class Personne
{
    public int Num { get; set; } = 99;
}
```

C#

Remarque

Pour générer `public int MyProperty { get; set; }` sous **Visual Studio Community 2022**

- écrire `prop`
- cliquer deux fois sur `tab`

C#

Le constructeur

- Une méthode particulière portant le nom de la classe et ne retournant aucune valeur.
- Toute classe en **C#** a un constructeur par défaut sans paramètre.
- Ce constructeur sans paramètre n'a aucun code.
- On peut le définir explicitement si un traitement est nécessaire (ou si on veut vérifier l'appel).
- La déclaration d'un objet de la classe (par exemple `Personne personne`) fait appel à ce constructeur sans paramètre.
- Toutefois, et pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs.

C#

Pour générer un constructeur sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Générer le constructeur`
- Sélectionner les paramètres du constructeur dans la liste (on choisit les trois attributs pour cet exemple)
- Valider

Nouveau contenu de la classe `Personne`

```
namespace CoursPoo
{
    class Personne
    {

        public int Num { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }

        public Personne(int num, string nom, string prenom)
        {
            Num = num;
            Nom = nom;
            Prenom = prenom;
        }

        public override string ToString()
        {
            return "Personne [num=" + Num + ", nom=" + Nom + ", prenom=
               " + Prenom + "]";
        }
    }
}
```

C#

En définissant ce constructeur avec trois paramètres, le constructeur par défaut (sans paramètre) n'existe plus, le `Main` suivant ne peut être exécuté

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                Nom = "wick",
                Prenom = "john",
                Num = 100
            };
            Console.WriteLine(personne);
            Personne personne2 = new Personne(200, "bob", "mike");
            Console.WriteLine(personne2);
            Console.ReadKey();
        }
    }
}
```

C#

Dans `Personne`, il faut donc redéfinir le constructeur sans paramètre

```
namespace CoursPoo
{
    class Personne
    {

        public int Num { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }
        public Personne(int num, string nom, string prenom)
        {
            Num = num;
            Nom = nom;
            Prenom = prenom;
        }

        public Personne()
        {
        }

        public override string ToString()
        {
            return "Personne [num=" + Num + ", nom=" + Nom + ", prenom=
                " + Prenom + "]";
```

C#

Maintenant, on peut utiliser les deux constructeurs dans `Main()`

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                Nom = "wick",
                Prenom = "john",
                Num = 100
            };
            Console.WriteLine(personne);
            Personne personne2 = new Personne(200, "bob", "mike");
            Console.WriteLine(personne2);
            Console.ReadKey();
        }
    }
}
```

C#

L'appel du constructeur à plusieurs paramètres peut aussi être simplifié [C# 9]

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                Nom = "wick",
                Prenom = "john",
                Num = 100
            };
            Console.WriteLine(personne);
            Personne personne2 = new(200, "bob", "mike");
            Console.WriteLine(personne2);
            Console.ReadKey();
        }
    }
}
```

C#

Dé-constructeur [**C# 10**]

- Une méthode particulière portant le même nom pour toutes les classes (`Deconstruct`) et ne retournant aucune valeur
- Il permet de retourner la valeur de certains attributs dans les paramètres `out` de la méthode

C#

Ajoutons le dé-constructeur suivant dans la classe `Personne`

```
public void Deconstruct(out int num, out string nom, out string prenom)
{
    num = Num;
    nom = Nom;
    prenom = Prenom;
}
```

C#

Pour tester

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                Nom = "wick",
                Prenom = "john",
                Num = 100
            };
            Personne personne2 = new Personne(200, "bob", "mike");
            (int n, string nom, string prenom) = personne2;
            Console.WriteLine($"{n} {nom} {prenom}");
            // affiche 200 bob mike
            Console.ReadKey();
        }
    }
}
```

C#

En testant l'égalité entre les deux objets suivants

```
Personne personne2 = new Personne(200, "bob", "mike");
Personne personne3 = new Personne(200, "bob", "mike");
Console.WriteLine(personne2.Equals(personne3));
```

© Achref EL MO

C#

En testant l'égalité entre les deux objets suivants

```
Personne personne2 = new Personne(200, "bob", "mike");
Personne personne3 = new Personne(200, "bob", "mike");
Console.WriteLine(personne2.Equals(personne3));
```

Le résultat est

```
False
```

C#

Explication

- Par défaut, deux objets d'une même classe sont égaux s'ils pointent sur le même espace mémoire (identiques).
- Pour comparer les valeurs, il faut redéfinir la méthode `Equals()` (qui peut utiliser `GetHashCode()`).
  - `Equals()` permet de définir comment tester sémantiquement l'égalité de deux objets.
  - `GetHashCode()` permet de retourner la valeur de hachage d'un objet (elle ne retourne pas un identifiant unique).
- `Equals()` et `GetHashCode()` doivent utiliser les mêmes attributs.

C#

Equals() et GetHashCode() doivent respecter les contraintes suivantes

- **Symétrie** : si `a.Equals(b)` retourne `true` alors `b.equals(a)` aussi.
- **Réflexivité** : `a.Equals(a)` retourne `true` si `a` est un objet non `null`.
- **Transitivité** : si `a.Equals(b)` et `b.Equals(c)` alors `a.Equals(c)`.
- **Consistance** : si `a.Equals(b)` alors `a.GetHashCode() == b.GetHashCode()`.
- `a.Equals(null)` retourne `false`.

C#

Génération sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Générer Equals et GetHashCode`
- Sélectionner les propriétés et valider

C#

Le code généré

```
public override bool Equals(object obj)
{
    return obj is Personne personne &&
        Num == personne.Num &&
        Nom == personne.Nom &&
        Prenom == personne.Prenom;
}

public override int GetHashCode()
{
    int hashCode = 487373800;
    hashCode = hashCode * -1521134295 + Num.GetHashCode();
    hashCode = hashCode * -1521134295 + EqualityComparer<string>.
       Default.GetHashCode(Nom);
    hashCode = hashCode * -1521134295 + EqualityComparer<string>.
       Default.GetHashCode(Prenom);
    return hashCode;
}
```

C#

En retestant le code précédent

```
Personne personne2 = new Personne(200, "bob", "mike");
Personne personne3 = new Personne(200, "bob", "mike");
Console.WriteLine(personne2.Equals(personne3));
```

© Achref EL MO

C#

En retestant le code précédent

```
Personne personne2 = new Personne(200, "bob", "mike");
Personne personne3 = new Personne(200, "bob", "mike");
Console.WriteLine(personne2.Equals(personne3));
```

Achref EL MO

Le résultat est

```
True
```

C#

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

© Achref EL MOUELHI ©

C#

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe `Personne`)

C#

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (le nombre d'objets instanciés de la classe `Personne`)

Définition

Un attribut dont la valeur est partagée par toutes les instances de la classe est appelée : attribut statique ou attribut de classe

C#

Exemple

- Si on veut créer un attribut contenant le nombre des objets créés à partir de la classe `Personne`
- Notre attribut doit être `static`, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut

C#

Ajoutons un attribut `static` appelé `NbrPersonnes` dans la classe `Personne`

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

© Achref EL MOUELHI ©

C#

Ajoutons un attribut `static` appelé `NbrPersonnes` dans la classe `Personne`

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

Incrémentons la valeur de `NbrPersonnes` dans les différents constructeurs de la classe `Personne`

```
public Personne(int num, string nom, string prenom)
{
    Num = num;
    Nom = nom;
    Prenom = prenom;
    NbrPersonnes++;
}

public Personne()
{
    NbrPersonnes++;
}
```

Testons cela dans le `Main`

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Personne.NbrPersonnes);
            Personne personne = new Personne()
            {
                Nom = "wick",
                Prenom = "john",
                Num = 100
            };
            Console.WriteLine(Personne.NbrPersonnes);
            Console.WriteLine(personne);
            Personne personne2 = new Personne(200, "bob", "mike");
            Console.WriteLine(Personne.NbrPersonnes);
            Console.WriteLine(personne2);
            Console.ReadKey();
        }
    }
}
```

C#

Exercice

- Définir une classe `Adresse` avec trois propriétés (`Rue`, `CodePostal` et `Ville` de type chaîne de caractère
- Définir un constructeur avec trois paramètres, un constructeur sans paramètre et la méthode `ToString()`
- Dans la classe `Personne`, ajouter une propriété `Adresses` de type tableau d'`Adresse` et définir un nouveau constructeur à quatre paramètres
- Dans le `Main`, créer trois objets, deux objets de type `Adresse` et un de type `Personne` et lui attribuer les adresses créées précédemment
- Afficher tous les attributs de cet objet de la classe `Personne`

C#

Opérateur de conversion [**C++ 11**]

- Une classe peut avoir plusieurs opérateurs de conversion : un opérateur au plus par type
- L'opérateur de conversion permet de réaliser une conversion personnalisée, implicite ou explicite, depuis ou vers un autre type
  - explicite : depuis un type quelconque vers un objet de classe qui définit l'opérateur
  - implicite : depuis la classe qui définit l'opérateur vers un autre type

C#

Considérons l'opérateur de conversion implicite suivant qui convertit un objet de type `Personne` en `int`

```
public static implicit operator int(Personne p)
{
    return p.Num;
}
```

© Achref EL

C#

Considérons l'opérateur de conversion implicite suivant qui convertit un objet de type `Personne` en `int`

```
public static implicit operator int(Personne p)
{
    return p.Num;
}
```

© Achref EL M

Testons cet opérateur dans le `Main`

```
int n = personne;

Console.WriteLine(n);
// affiche 100
```

C#

Considérons l'opérateur de conversion explicite suivant qui convertit un `int` en objet de type `Personne`

```
public static implicit operator int(Personne p)
{
    return p.Num;
}
```

© Achref EL M

C#

Considérons l'opérateur de conversion explicite suivant qui convertit un `int` en objet de type `Personne`

```
public static implicit operator int(Personne p)
{
    return p.Num;
}
```

© Achref EL M

Testons cet opérateur dans le `Main`

```
Personne pers = (Personne)600;

Console.WriteLine($"{pers.Num} {pers.Nom}");
// affiche 600 doe
```

C#

Chaînage optionnel (?.)

Permet d'arrêter immédiatement l'exécution d'une expression si on rencontre la valeur `null`.

© Achref EL MOUELHI ©

C#

Chaînage optionnel (?.)

Permet d'arrêter immédiatement l'exécution d'une expression si on rencontre la valeur `null`.

Problématique

```
Personne personne5 = null;
int x = 2;
if (x == 0)
{
    personne5 = new Personne(11, "Hanks", "Tom");
}
Console.WriteLine(personne5.Nom);
// Exception levée
```

Première solution

```
Personne personne5 = null;
int x = 2;
if (x == 0)
{
    personne5 = new Personne(11, "Hanks", "Tom");
}
if (personne5 != null)
{
    Console.WriteLine(personne5.Nom);
}
```

© Achref EL MOUELHI

Première solution

```
Personne personne5 = null;
int x = 2;
if (x == 0)
{
    personne5 = new Personne(11, "Hanks", "Tom");
}
if (personne5 != null)
{
    Console.WriteLine(personne5.Nom);
}
```

Deuxième solution avec le chaînage optionnel

```
Personne personne5 = null;
int x = 2;
if (x == 0)
{
    personne5 = new Personne(11, "Hanks", "Tom");
}
Console.WriteLine(personne5?.Nom);
```

Première solution

```
Personne personne5 = null;
int x = 2;
if (x == 0)
{
    personne5 = new Personne(11, "Hanks", "Tom");
}
if (personne5 != null)
{
    Console.WriteLine(personne5.Nom);
}
```

Deuxième solution avec le chaînage optionnel

```
Personne personne5 = null;
int x = 2;
if (x == 0)
{
    personne5 = new Personne(11, "Hanks", "Tom");
}
Console.WriteLine(personne5?.Nom);
```

Une expression contenant un chaînage optionnel ne peut être à gauche d'une affectation.

C#

Définition

- Concept **C#** qui facilite l'accès à un tableau d'objet défini dans un objet.
- (Autrement dit) Utiliser une "classe" comme un tableau.

C#

Considérons la classe `ListePersonnes` suivante qui contient un tableau de `Personne`

```
class ListePersonnes
{
    private Personne[] Personnes;
}
```

Comment faire pour enregistrer des personnes dans le tableau `personnes` et les récupérer facilement en faisant `listePersonnes[i]`

```
ListePersonnes mesAmis = new ListePersonnes(2);

mesAmis[0] = personne;
mesAmis[1] = personne2;
```

© Achref EL MOU

C#

Contenu de la classe `ListePersonnes`

```
namespace CoursPoo
{
    class ListePersonnes
    {
        public Personne[] Personnes { get; set; }
        public int NbrPersonnes { get; set; }

        public ListePersonnes(int i)
        {
            Personnes = new Personne[i];
            NbrPersonnes = 0;
        }
    }
}
```

C#

Ajoutons l'indexeur à la classe `ListePersonnes`

```
public Personne this[int i]
{
    get { return Personnes[i]; }
    set { Personnes[i] = value; NbrPersonnes++; }
}
```

Explication

- Pour mieux comprendre, remplacer `this` par `ListePersonnes` dans `public Personne this[int i]`, ça devient `public Personne ListePersonnes[int i]`
- Donc, c'est comme-ci on définit comment utiliser la classe `ListePersonnes` comme un tableau.

C#

Pour tester

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne personne = new Personne()
            {
                Nom = "wick",
                Prenom = "john",
                Num = 100
            };
            Personne personne2 = new Personne(200, "bob", "mike");
            ListePersonnes mesAmis = new ListePersonnes(2);
            mesAmis[0] = personne;
            mesAmis[1] = personne2;
            for (int i = 0; i < mesAmis.NbrPersonnes; i++)
            {
                Console.WriteLine("mesAmis[{0}] : {1}", i, mesAmis[i]);
            }
            Console.ReadKey();
        }
    }
}
```

C#

Exercice

- Dans `Personne`, définir un indexeur sur les adresses
- Dans le `Main`, vérifier qu'il est possible d'accéder à l'adresse d'une personne de `ListePersonnes` en faisant par exemple `mesAmis[0][0]`

C#

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une [sorte de]** `Classe2`

C#

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire

C#

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau

C#

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne

C#

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom

C#

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom
- Donc, on peut mettre en commun les attributs numéro, nom et prénom dans une classe `Personne`

C#

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom
- Donc, on peut mettre en commun les attributs numéro, nom et prénom dans une classe `Personne`
- Les classes `Étudiant` et `Enseignant` hériteront de la classe `Personne`

C#

Particularité du **C#**

- L'héritage multiple n'est pas autorisé
- C'est-à-dire, une classe ne peut hériter de plusieurs classes différentes

C#

Contenu de la classe `Enseignant`

```
namespace CoursPoo
{
    class Enseignant : Personne
    {
    }
}
```

© Achref EL MOU

C#

Contenu de la classe `Enseignant`

```
namespace CoursPoo
{
    class Enseignant : Personne
    {
    }
}
```

Contenu de la classe `Etudiant`

```
namespace CoursPoo
{
    class Etudiant : Personne
    {
    }
}
```

C#

Ensuite

- Créer un attribut `niveau` dans la classe `Etudiant` puis générer les getter et setter
- Créer un attribut `salaire` dans la classe `Enseignant` puis générer les getter et setter

C#

Contenu de la classe `Enseignant`

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

© Achref EL MOU

C#

Contenu de la classe `Enseignant`

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

Contenu de la classe `Etudiant`

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

C#

Pour créer un objet de type `Enseignant`

```
Enseignant enseignant = new Enseignant();
enseignant.Num = 300;
enseignant.Nom = "green";
enseignant.Prenom = "jonas";
enseignant.Salaire = 1700;
Console.WriteLine(enseignant);
```

© Achref EL MOU

C#

Pour créer un objet de type `Enseignant`

```
Enseignant enseignant = new Enseignant();
enseignant.Num = 300;
enseignant.Nom = "green";
enseignant.Prenom = "jonas";
enseignant.Salaire = 1700;
Console.WriteLine(enseignant);
```

En exécutant, le résultat est :

```
Personne [num=300, nom=green, prenom=jonas]
```

C#

Pour créer un objet de type `Enseignant`

```
Enseignant enseignant = new Enseignant();
enseignant.Num = 300;
enseignant.Nom = "green";
enseignant.Prenom = "jonas";
enseignant.Salaire = 1700;
Console.WriteLine(enseignant);
```

En exécutant, le résultat est :

```
Personne [num=300, nom=green, prenom=jonas]
```

Mais on ne voit pas le salaire, pourquoi ?

Comme on n'a pas redéfini la méthode `ToString()`, on a utilisé celle de la classe mère

C#

Et si on génère le `ToString()` dans la classe `Enseignant`

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

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

© Achref EL MOUELHI

C#

Et si on génère le `ToString()` dans la classe `Enseignant`

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

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

Modifions son contenu

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

© Aghiles EL MOUELHI

Définissons un constructeur à quatre paramètres dans `Enseignant`

```
namespace CoursPoo
{
    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
              + "]";
        }
    }
}
```

Faisons la même chose dans `Etudiant`

```
namespace CoursPoo
{
    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 + "]
                ";
        }
    }
}
```

C#

On peut créer un objet de la classe `Enseignant` ainsi

```
Enseignant enseignant2 = new Enseignant(400, "rubi", "guillaume", 900);
```

© Achref EL MOUELHI ©

C#

On peut créer un objet de la classe `Enseignant` ainsi

```
Enseignant enseignant2 = new Enseignant(400, "rubi", "guillaume", 900);
```

Ou ainsi

```
Personne enseignant2 = new Enseignant(400, "rubi", "guillaume", 900);
```

© Achref El MOUELHI ©

C#

On peut créer un objet de la classe `Enseignant` ainsi

```
Enseignant enseignant2 = new Enseignant(400, "rubi", "guillaume", 900);
```

Ou ainsi

```
Personne enseignant2 = new Enseignant(400, "rubi", "guillaume", 900);
```

© Achref El MOUELHI ©

Ceci est faux

```
Enseignant enseignant2 = new Personne(400, "rubi", "guillaume");
```

C#

Remarque

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

© Achref EL MOUELHI ©

C#

Remarque

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

EL MOUELHI©

Exemple

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

C#

Exercice

1. Créer trois objets : un premier de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne`
2. Stocker les tous dans un seul tableau `personnes`
3. Parcourir le tableau et afficher pour chacun soit le `num` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

© Achref EL

C#

Exercice

1. Créer trois objets : un premier de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne`
2. Stocker les tous dans un seul tableau `personnes`
3. Parcourir le tableau et afficher pour chacun soit le `num` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

© Achref EL

Pour parcourir un tableau, on peut utiliser le raccourci de `foreach`

```
Personne[] personnes = { personne, etudiant, enseignant };
foreach (Personne perso in personnes)
{
    Console.WriteLine(perso);
}
```

C#

Solution 1

```
foreach (Personne perso in personnes)
{
    if (perso is Enseignant)
    {
        Console.WriteLine(((Enseignant) perso).Salaire);
    }
    else if (perso is Etudiant)
    {
        Console.WriteLine(((Etudiant)perso).Niveau);
    }
    else
    {
        Console.WriteLine(perso.Num);
    }
}
```

C#

Solution 2

```
foreach (Personne perso in personnes)
{
    if (perso is Etudiant)
    {
        Console.WriteLine((perso as Etudiant).Niveau);
    }
    else if (perso is Enseignant)
    {
        Console.WriteLine((perso as Enseignant).Salaire);
    }
    else
    {
        Console.WriteLine(perso.Num);
    }
}
```

C#

Solution 3 [C# 7.0]

```
foreach (Personne perso in personnes)
{
    if (perso is Etudiant et)
    {
        Console.WriteLine(et.Niveau);
    }
    else if (perso is Enseignant ens)
    {
        Console.WriteLine(ens.Salaire);
    }
    else
    {
        Console.WriteLine(perso.Num);
    }
}
```

C#

Polymorphisme en **C#**

- Polymorphisme : prendre plusieurs formes
- Comment une méthode peut être redéfinie de plusieurs façons différentes
- Utilisation des mots clés `new`, `virtual` et `override`

C#

Commençons par définir une méthode `AfficherDetails()` **dans** `Personne`

```
public void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom } ");
}
```

C#

Redéfinissons la méthode `AfficherDetails()` dans `Etudiant`

```
public void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom } { Niveau }");
}
```

© Achref EL MOU

C#

Redéfinissons la méthode `AfficherDetails()` **dans** `Etudiant`

```
public void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom } { Niveau }");
}
```

Testons la méthode `afficherDetails()`

```
Etudiant etudiant2 = new Etudiant(500, "bernhardt",
   "céline", "ingénieur");
etudiant2.AfficherDetails();

// affiche céline bernhardt ingénieur
```

C#

La méthode est soulignée en vert et un message nous propose d'ajouter le mot clé `new`

© Achref EL MOUELHI ©

C#

La méthode est soulignée en vert et un message nous propose d'ajouter le mot clé `new`

Le mot clé `new` peut être ajouté avant ou après `public`

```
public new void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom } { Niveau }");
}
```

© Achref El

C#

La méthode est soulignée en vert et un message nous propose d'ajouter le mot clé `new`

Le mot clé `new` peut être ajouté avant ou après `public`

```
public new void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom } { Niveau }");
}
```

Testons la méthode `afficherDetails()`

```
Etudiant etudiant2 = new Etudiant(500, "bernhardt", "céline", "ingé
   nieur");
etudiant2.AfficherDetails();

// affiche céline bernhardt ingénieur
```

C#

Modifions le type de l'objet `etudiant2`

```
Personne etudiant2 = new Etudiant(500, "bernhardt", "céline", "
   ingénieur");
etudiant2.AfficherDetails();

// affiche céline bernhardt
```

© Achref EL MOUEL

C#

Modifions le type de l'objet `etudiant2`

```
Personne etudiant2 = new Etudiant(500, "bernhardt", "céline", "
   ingénieur");
etudiant2.AfficherDetails();

// affiche céline bernhardt
```

Comment résoudre ce problème ?

- Remplacer le mot-clé `new` par `override`
- Ajouter le mot-clé `virtual` à la signature de la méthode `AfficherDetails()` dans `Personne`
- Les mots-clés `override` ne peut être utilisé sans `virtual`

C#

Modifions la méthode `AfficherDetails()` dans `Personne`

```
public virtual void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom }");
}
```

Modifions la méthode `AfficherDetails()` dans `Etudiant`

```
public override void AfficherDetails()
{
    Console.Write($"{ Prenom } { Nom } { Niveau }");
}
```

Les mots-clés `virtual` et `override` peuvent aussi être ajoutés avant ou après `public`

C#

Pour tester

```
Personne etudiant2 = new Etudiant(500, "bernhardt",
   "céline", "ingénieur");
etudiant2.AfficherDetails();

// affiche céline bernhardt ingénieur
```

C#

Classe abstraite

- Classe qu'on ne peut instancier.
- Déclarée avec le mot-clé `abstract`,

© Achref EL MOUELHI ©

C#

Classe abstraite

- Classe qu'on ne peut instancier.
- Déclarée avec le mot-clé `abstract`,

Si on déclare la classe `Personne` abstraite

```
public abstract class Personne
{
    ...
}
```

EL MOUELHI ©

C#

Classe abstraite

- Classe qu'on ne peut instancier.
- Déclarée avec le mot-clé `abstract`,

Si on déclare la classe `Personne` abstraite

```
public abstract class Personne
{
    ...
}
```

Ce code sera souligné en rouge

```
Personne personne2 = new Personne(2, "bob", "mike");
```

C#

Méthode abstraite

- Méthode non-implémentée (sans code) lors de sa déclaration,
- Déclarée dans une classe abstraite,
- Méthode implémentée par les classes filles de la classe abstraite.

© Achref EL MOUELHI

C#

Méthode abstraite

- Méthode non-implémentée (sans code) lors de sa déclaration,
- Déclarée dans une classe abstraite,
- Méthode implémentée par les classes filles de la classe abstraite.

© Achref EL MOUELHI ©

Déclarons une méthode abstraite `AfficherType()` **dans** `Personne`

```
public abstract void AfficherType();
```

C#

Méthode abstraite

- Méthode non-implémentée (sans code) lors de sa déclaration,
- Déclarée dans une classe abstraite,
- Méthode implémentée par les classes filles de la classe abstraite.

Déclarons une méthode abstraite `AfficherType()` **dans** `Personne`

```
public abstract void AfficherType();
```

Remarque

Suite à la déclaration de `AfficherType()` dans `Personne`, les deux classes `Etudiant` et `Enseignant` sont soulignées en rouge.

Pour générer les méthodes abstraites sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe (`Etudiant` ou `Enseignant`) dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Implémenter une classe abstraite`

© Achref EL MOUELHI ©

Pour générer les méthodes abstraites sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe (`Etudiant` ou `Enseignant`) dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Implémenter une classe abstraite`

© MOUELHI

Code généré

```
public override void AfficherType()
{
    throw new NotImplementedException();
}
```

Pour générer les méthodes abstraites sous **Visual Studio Community 2022**

- Faire un clic droit sur le nom de la classe (`Etudiant` ou `Enseignant`) dans l'éditeur de code
- Choisir `Actions rapides et refactorisations`
- Cliquer sur `Implémenter une classe abstraite`

© R. MOUELHI ©

Code généré

```
public override void AfficherType()
{
    throw new NotImplementedException();
}
```

Remplaçons le code généré par le suivant

```
public override void AfficherType()
{
    Console.WriteLine("Enseignant");
}
```

C#

Testons la méthode `AfficherType()` dans `Main`

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Enseignant enseignant2 = new Enseignant(400, "rubi", "
               guillaume", 1700);
            enseignant2.AfficherType();
            Console.ReadKey();
        }
    }
}
```

C#

Classe `sealed` (fermée)

Une classe est déclarée `sealed` si on ne peut l'hériter.

© Achref EL MOUELHI ©

C#

Classe `sealed` (fermée)

Une classe est déclarée `sealed` si on ne peut l'hériter.

Si on déclare la classe `Enseignant` sellée

```
public sealed class Enseignant : Personne
{
    ...
}
```

© Ach HI ©

C#

Classe `sealed` (fermée)

Une classe est déclarée `sealed` si on ne peut l'hériter.

Si on déclare la classe `Enseignant` sellée

```
public sealed class Enseignant : Personne
{
    ...
}
```

On ne peut créer une classe `Doctorant` qui hérite de `Enseignant` ('Doctorant' : dérivation du type sealed 'Enseignant' impossible)

```
class Doctorant : Enseignant
{
    ...
}
```

C#

Méthode `sealed`

Une méthode est déclarée `sealed` si on ne peut la redéfinir.

© Achref EL MOUELHI ©

C#

Méthode `sealed`

Une méthode est déclarée `sealed` si on ne peut la redéfinir.

Déclarons une méthode sellée dans la classe `Enseignant` (Supprimons le mot-clé `sealed` de la déclaration de la classe `Enseignant`)

```
public  class Enseignant : Personne
{
    // code précédent
    public sealed override void AfficherType()
    {
        Console.WriteLine("Enseignant");
    }
}
```

EL MOUELHI ©

C#

La méthode `AfficherType` ne peut être redéfinie dans `Doctorant` car elle est déclarée `sealed` dans `Enseignant`

```
namespace CoursPoo
{
    class Doctorant : Enseignant
    {
        public override void AfficherType()
        {
            Console.WriteLine("Doctorant");
        }
    }
}
```

C#

Exemple de classe scellée prédéfinie en **C#**

```
string
```

© Achref EL MOUELHI

C#

Exemple de classe scellée prédéfinie en **C#**

```
string
```

Pourquoi `string` est scellée ?

- Garantir immuabilité,
- Interdire d'avoir des sous-classes de `string` qui ne seront pas immuables.

C#

Pourquoi `string` est immuable ?

- **Sécurité** : les paramètres de connexions réseau, les URL de connexion à la base de données, les noms d'utilisateur/mots de passe sont souvent représentées par des `string`. Rendre la classe scellée $\Rightarrow$ impossible de modifier ces paramètres.
- **Synchronisation et concurrence** : rendre `string` immuable $\Rightarrow$ **thread-safe** : pas de problèmes de synchronisation.
- ...

C#

Héritage en **C#**

- une classe peut hériter d'une seule classe
- une classe peut implémenter plusieurs interfaces

C#

Interface en **C#**

- est déclarée avec le mot clé `interface`
- ne contient pas de champs
- contient seulement les signatures de méthodes sans les implémenter

© Achref EL Mo

C#

Interface en **C#**

- est déclarée avec le mot clé `interface`
- ne contient pas de champs
- contient seulement les signatures de méthodes sans les implémenter

Interface, classe et instanciation

- une interface ne peut être instanciée
- une classe peut implémenter plusieurs interfaces
- une interface peut implémenter une (ou plusieurs) autre(s) interface(s)

C#

Pour créer une interface sous **Visual Studio Community 2022**

- Faire clic droit sur le nom du projet dans l'`Explorateur de solutions`
- Aller dans `Ajouter > Nouvel élément`
- Choisir `Interface`
- Saisir `ISalutation` dans `Nom` et valider

Créer une interface

```
interface ISalutation
{
    void DireBonjour();
}
```

© Achref EL MOUELHI ©

Créer une interface

```
interface ISalutation
{
    void DireBonjour();
}
```

Implémenter une interface

```
public class Etudiant : Personne, ISalutation
{
```

© Achref EL MOUELHI ©

Créer une interface

```
interface ISalutation
{
    void DireBonjour();
}
```

Implémenter une interface

```
public class Etudiant : Personne, ISalutation
{
```

Implémenter implicitement les méthodes de l'interface dans `Personne`

```
public void DireBonjour()
{
    Console.WriteLine($"Bonjour {Prenom} {Nom}");
}
```

© Achref EL MOUELHI ©

Créer une deuxième interface

```
interface IGreeting
{
    void SayHello();
}
```

© Achref EL MOUELHI ©

Créer une deuxième interface

```
interface IGreeting
{
    void SayHello();
}
```

Implémenter plusieurs interfaces

```
public class Etudiant : Personne, ISalutation, IGreeting
{
}
```

© ACH QUELHI ©

Créer une deuxième interface

```
interface IGreeting
{
    void SayHello();
}
```

Implémenter plusieurs interfaces

```
public class Etudiant : Personne, ISalutation, IGreeting
{
}
```

Implémenter explicitement les méthodes de l'interface dans `Personne`

```
void IGreeting.SayHello()
{
    Console.WriteLine($"Hello { Prenom } { Nom }");
}
```

C#

Appeler la méthode `DireBonjour()`

```
Etudiant etudiant2 = new Etudiant(500, "bernhardt", "céline", "ingénieur");

etudiant2.DireBonjour();
// affiche Bonjour céline bernhardt

// on ne trouve pas directement SayHello()
((IGreeting)etudiant2).SayHello();
// affiche Hello céline bernhardt
```

© Achref EL MOU

C#

Appeler la méthode `DireBonjour()`

```
Etudiant etudiant2 = new Etudiant(500, "bernhardt", "céline", "ingénieur");

etudiant2.DireBonjour();
// affiche Bonjour céline bernhardt

// on ne trouve pas directement SayHello()
((IGreeting)etudiant2).SayHello();
// affiche Hello céline bernhardt
```

Appeler la méthode `SayHello()`

```
IGreeting etudiant3 = new Etudiant(500, "maggio", "candice", "docteur");

etudiant3.SayHello();
// affiche Hello candice maggio

// on ne trouve pas directement DireBonjour()
((Etudiant)etudiant3).DireBonjour();
// affiche Bonjour candice maggio
```

Si par exemple `IGreeting` implémente une autre interface

```
interface IGreeting : IMainMethod
{
    void SayHello();
}
```

L'interface `IMainMethod`

```
interface IMainMethod
{
    void SayAnything();
}
```

MOUELHI ©

Si par exemple `IGreeting` implémente une autre interface

```
interface IGreeting : IMainMethod
{
    void SayHello();
}
```

L'interface `IMainMethod`

```
interface IMainMethod
{
    void SayAnything();
}
```

Dans ce cas, la classe `Etudiant` doit aussi implémenter la méthode `SayAnything` de l'interface `IMainMethod`

MOUELHI ©

C#

Et si on veut parcourir les objets `Personne` de la classe `ListePersonnes` comme si c'était une vraie liste, c'est-à-dire :

```
foreach(Personne personne in mesAmis)
{
    personne.DireBonjour();
}
```

Un message d'erreur nous informe qu'il faut avoir une définition publique de `GetEnumerator()`.

C#

Et si on veut parcourir les objets `Personne` de la classe `ListePersonnes` comme si c'était une vraie liste, c'est-à-dire :

```
foreach(Personne personne in mesAmis)
{
    personne.DireBonjour();
}
```

Un message d'erreur nous informe qu'il faut avoir une définition publique de `GetEnumerator()`.

On va donc implémenter l'interface générique `IEnumerable <T>` qui a la méthode `GetEnumerator()`.

C#

Implémentons l'interface `IEnumerable`

```
class ListePersonnes : IEnumerable<Personne>
```

© Achref EL MOUELHI ©

C#

Implémentons l'interface `IEnumerable`

```
class ListePersonnes : IEnumerable<Personne>
```

Code souligné en rouge ?

© Achref EL MOUELHI ©

C#

Implémentons l'interface `IEnumerable`

```
class ListePersonnes : IEnumerable<Personne>
```

Code souligné en rouge ?

Achref EL M... HI ©

Quoi faire ?

- faire clic droit sur `IEnumerable` et choisir `Actions rapides et refactorisations`
- cliquer ensuite sur `Implémenter l'interface via 'Personnes'`

C#

Code généré

```
// implémentation implicite de l'interface  IEnumerable
public IEnumerator<Personne> GetEnumerator()
{
    return ((IEnumerable<Personne>)Personnes).GetEnumerator();
}

// implémentation explicite de l'interface  IEnumerable
IEnumerator IEnumerable.GetEnumerator()
{
    return Personnes.GetEnumerator();
}
```

C#

Maintenant, ce code est exécutable et n'est plus souligné en rouge

```
foreach(Personne personne in mesAmis)
{
    personne.DireBonjour();
}
```

© Achref EL MOUELHI

C#

Maintenant, ce code est exécutable et n'est plus souligné en rouge

```
foreach(Personne personne in mesAmis)
{
    personne.DireBonjour();
}
```

IEnumerable : autres propriétés

- On a utilisé `IEnumerable` pour énumérer les objets d'une classe comme si cette dernière était un tableau (itérer sur une classe)
- On peut également l'utiliser avec une méthode pour retourner plusieurs variables (une valeur chaque fois qu'on l'appelle, itérer sur une méthode)

C#

Considérons la méthode `Power` suivante

```
public static int Power(int x, int n)
{
    int result = 1;
    for (int i = 0; i < n; i++)
    {
        result = result * x;
    }
    return result;
}
```

© Acm

On appelle cette méthode

```
Console.Write($"{ Power(2, 8) });
// affiche 256
Console.ReadKey();
```

C#

Et si on voulait que cette méthode nous retourne tous les exposants de `x` jusqu'au `n` ?

© Achref EL MOUELHI ©

C#

Et si on voulait que cette méthode nous retourne tous les exposants de `x` jusqu'au `n` ?

Dans ce cas, `Power` doit retourner un `IEnumerable`.

© Achref EL M... HI ©

C#

Et si on voulait que cette méthode nous retourne tous les exposants de `x` jusqu'au `n` ?

Dans ce cas, `Power` doit retourner un `IEnumerable`.

Et il faut utiliser le mot clé `yield` avec `return`.

© Achref EL MOUELHI ©

C#

Et si on voulait que cette méthode nous retourne tous les exposants de `x` jusqu'au `n` ?

Dans ce cas, `Power` doit retourner un `IEnumerable`.

Et il faut utiliser le mot clé `yield` avec `return`.

`yield` permet de retourner plusieurs éléments un par un.

© Achref EL MOUELHI ©

La méthode `Power` devient

```
public static IEnumerable<int> Power(int x, int n)
{
    int result = 1;
    for (int i = 0; i < n; i++)
    {
        result = result * x;
        yield return result;
    }
}
```

© Achref EL MOU

La méthode `Power` devient

```
public static IEnumerable<int> Power(int x, int n)
{
    int result = 1;
    for (int i = 0; i < n; i++)
    {
        result = result * x;
        yield return result;
    }
}
```

Maintenant on peut itérer sur la méthode

```
foreach (int elt in  Power(2, 8))
{
    Console.Write("{0} ", elt);
}
Console.ReadKey();
```

La méthode `Power` devient

```
public static IEnumerable<int> Power(int x, int n)
{
    int result = 1;
    for (int i = 0; i < n; i++)
    {
        result = result * x;
        yield return result;
    }
}
```

Maintenant on peut itérer sur la méthode

```
foreach (int elt in  Power(2, 8))
{
    Console.Write("{0} ", elt);
}
Console.ReadKey();
```

et ça affiche 2 4 8 16 32 64 128 256

C#

Exercice

En utilisant `yield` et `IEnumerable`, écrire un programme C# qui :

- demande à l'utilisateur de saisir un entier positif
- affiche tous les nombres premiers inférieurs à cet entier positif

© Achref EL MOU

C#

Exercice

En utilisant `yield` et `IEnumerable`, écrire un programme C# qui :

- demande à l'utilisateur de saisir un entier positif
- affiche tous les nombres premiers inférieurs à cet entier positif

Exemple de `Main`

```
var nbr = int.Parse(Console.ReadLine());
foreach(var elt in GetAllPremier(nbr))
{
    Console.WriteLine(elt);
}
Console.ReadKey();
```

Solution

```
public static bool EstPremier(int n)
{
    for (int i = 2; i < n; i++)
    {
        if (n % i == 0)
            return false;
    }
    return true;
}

public static IEnumerable<int> GetAllPremier(int n)
{
    for(int i = 1; i <= n; i++)
    {
        if (EstPremier(i))
            yield return i;
    }
}
```

C#

Classe partielle

Une classe est déclarée partielle (avec le mot clé `partial`) si son code peut être fractionnée sur plusieurs fichiers

© Ach

Exemple : dans un fichier `Vehicule1.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public string Modele { get; set; }
    }
}
```

© Achref EL MOUELHI

Exemple : dans un fichier `Vehicule1.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public string Modele { get; set; }
    }
}
```

Exemple : dans un fichier `Vehicule2.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public string Marque { get; set; }
    }
}
```

Exemple : dans un fichier `Vehicule1.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public string Modele { get; set; }
    }
}
```

Exemple : dans un fichier `Vehicule2.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public string Marque { get; set; }
    }
}
```

On ne peut déclarer un attribut et/ou une méthode avec le même nom dans les deux fichiers.

Rien ne change pour l'instanciation

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Vehicule vehicule = new Vehicule()
            {
                Marque = "peugeot",
                Modele = "208"
            };
            Console.WriteLine($"{ vehicule.Marque } : { vehicule.Modele
                }");
            // affiche peugeot : 208
            Console.ReadKey();
        }
    }
}
```

On peut donc instancier la classe comme si elle est défini dans un seul fichier

C#

Si on crée un constructeur à deux paramètres dans `Vehicule1.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public Vehicule(string modele, string marque)
        {
            Modele = modele;
            Marque = marque;
        }

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

C#

Si on crée un constructeur à deux paramètres dans `Vehicule1.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public Vehicule(string modele, string marque)
        {
            Modele = modele;
            Marque = marque;
        }

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

Le constructeur par défaut n'existe plus, donc le code du `Main` sera souligné en rouge.

C#

On peut définir un deuxième constructeur sans paramètre dans `Vehicule2.cs`

```
namespace CoursPoo
{
    partial class Vehicule
    {
        public Vehicule()
        {
        }
        public string Marque { get; set; }
    }
}
```

C#

On peut donc utiliser les deux constructeurs

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Vehicule vehicule = new Vehicule()
            {
                Marque = "peugeot",
                Modele = "208"
            };
            Console.WriteLine($"{ vehicule.Marque } : { vehicule.Modele
                }");
            Vehicule vehicule2 = new Vehicule("ford", "kuga");
            Console.WriteLine($"{ vehicule2.Marque } : { vehicule2.
              Modele }");
            Console.ReadKey();
        }
    }
}
```

C#

Méthode partielle

- Déclarée avec le mot clé `partial`
- Sa déclaration est définie dans une classe partielle
- Son implémentation est définie dans une autre classe partielle.

C#

Dans le fichier `Class1.cs`

```
partial void DireBonjour()
{
    Console.WriteLine($"Bonjour {Nom}");
}
```

© Achref EL

Dans le fichier `Class2.cs`

```
partial void DireBonjour();
```

C#

Règles pour les méthodes partielles

- Les signatures des deux méthodes partielles doivent être identiques.
- La méthode ne doit pas avoir de valeur de retour (`void`).
- Aucun niveau de visibilité, autre que private, n'est autorisé.

C#

De la même manière qu'une classe partielle

une interface peut aussi être déclarée partielle.

© Achren - H&H ©

C#

Opérateur

- **C#** nous offre la possibilité de surcharger les opérateurs
  - arithmétiques : +, -, *, /, %, =, +=, ++...
  - ou de comparaison : ==, !=, >=, <=...
- Il est possible de définir comment utiliser ces opérateurs entre deux objets

C#

Considérons la classe `Fraction` suivante

```
namespace CoursPoo
{
    internal class Fraction
    {
        public Fraction(int numerateur, int denominateur)
        {
            Numerateur = numerateur;
            Denominateur = denominateur;
        }

        public int Numerateur { get; set; }
        public int Denominateur { get; set; }

        public override string ToString()
        {
             return $"{Numerateur} / {Denominateur}";
        }
    }
}
```

C#

Objectif

- calculer l'opposé d'une fraction (opérateur un-aire -)
- calculer la somme de deux fractions (opérateur binaire +)
- comparer deux fractions (opérateur == ou !=)

© Achref El

C#

Objectif

- calculer l'opposé d'une fraction (opérateur un-aire -)
- calculer la somme de deux fractions (opérateur binaire +)
- comparer deux fractions (opérateur == ou !=)

© Achref El

Opérateur en **C#**

méthode `static` déclarée avec le mot-clé `opérator`

Considérons la classe `Fraction` suivante

```
namespace CoursPoo
{
    class Fraction
    {
        public Fraction(int numerateur, int denominateur)
        {
            Numerateur = numerateur;
            Denominateur = denominateur;
        }

        public int Numerateur { get; set; }
        public int Denominateur { get; set; }

        public override string ToString()
        {
             return $"{Numerateur} / {Denominateur}";
        }

        public static Fraction operator +(Fraction a, Fraction b)
        {
             return new Fraction(a.Numerateur * b.Denominateur + b.Numerateur * a.Denominateur,
                 a.Denominateur * b.Denominateur);
        }

        public static Fraction operator -(Fraction a)
        {
             return new Fraction(-a.Numerateur, a.Denominateur);
        }
    }
}
```

C#

Pour tester

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Fraction a = new Fraction(1, 3);
            Console.WriteLine(a);
            Fraction b = new Fraction(2, 6);
            Console.WriteLine(b);
            Console.WriteLine($"somme : { a + b }");
            Console.WriteLine($"opposé : { -a }");
            Console.WriteLine($"somme + opposé : { b + (-a) }");
            Console.ReadKey();
        }
    }
}
```

C#

Remarque

- Les opérateurs de comparaison suivants doivent être surchargés par paires.
  - == et !=
  - > et <
  - >= et <=
- Si l'un des opérateurs d'une paire est surchargé, l'autre doit également l'être pour garantir la cohérence.
- Les opérateurs combinés comme +=, −= ne peuvent pas être surchargés directement, mais ils sont implicites si on surcharge (+, −...).
- `new` et `delete` ne sont pas surchargeables en **C#**, contrairement à **C++**.
- Les opérateurs d'accès aux membres (., ->) ne peuvent pas non plus être surchargés.

C#

Ajoutons les deux opérateurs de comparaison suivants

```
public static bool operator ==(Fraction a, Fraction b)
{
    return a.Numerateur / b.Numerateur == a.Denominateur / b.Denominateur;
}

public static bool operator !=(Fraction a, Fraction b)
{
    return a.Numerateur / b.Numerateur != a.Denominateur / b.Denominateur;
}
```

C#

Pour tester

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Fraction a = new Fraction(1, 3);
            Console.WriteLine(a);
            Fraction b = new Fraction(2, 6);
            Console.WriteLine(b);
            Console.WriteLine($"égalité : { a == b }");
            Console.WriteLine($"inégalité : { a != b }");
            Console.ReadKey();
        }
    }
}
```

C#

Structure en **C#**

- déclaré avec le mot-clé `struct`
- vieux concept connu en langage **C** et **C++**
- permettant d'avoir plusieurs données (aucune contrainte sur les types) / méthodes au sein d'un seul composant.

© Achref EL M

C#

Structure en **C#**

- déclaré avec le mot-clé `struct`
- vieux concept connu en langage **C** et **C++**
- permettant d'avoir plusieurs données (aucune contrainte sur les types) / méthodes au sein d'un seul composant.

On peut définir une structure dans

- un espace de nom
- une classe
- une autre structure

C#

Pour créer une structure sous **Visual Studio Community 2022**

- Clic droit sur le nom du projet dans l'`Explorateur de solutions`
- Aller dans `Ajouter > Nouvel élément`
- Dans la rubrique `Code`, Choisir `Fichier de code`
- Saisir le nom dans `Nom :` et valider

C#

Une première structure

```
namespace CoursPoo
{
    public struct Livre
    {
        public string isbn;
        public string titre;
        public int nbrPages;

        public void AfficherDetails()
        {
            Console.WriteLine($" Isbn : {isbn} \n Titre {
               titre} \n Nombre de pages : {nbrPages}");
        }
    }
}
```

C#

Instancier une structure = instancier une classe

```
namespace CoursPoo
{
    class Program
    {
        static void Main(string[] args)
        {
            Livre livre = new Livre();
            livre.titre = "programmation C#";
            livre.isbn = "1111111111";
            livre.nbrPages = 1000;
            livre.AfficherDetails();

            Console.ReadKey();
        }
    }
}
```

C#

Classe Vs Structure : quelle différence alors ?

- Passage par référence pour les classes, passage par valeur pour les structures
- Les classes supportent l'héritage, les structures non.
- Les structures n'acceptent pas la valeur `null`
- ...

C#

Qu'affiche le programme suivant ?

```
static void Main(string[] args)
{
    Livre livre = new Livre();
    livre.titre = "programmation C#";
    livre.isbn = "1111111111";
    livre.nbrPages = 1000;
    livre.AfficherDetails();

    Livre livre2 = livre;
    livre2.titre = "Struct C#";
    livre2.isbn = "222222222";
    livre2.nbrPages = 200;

    Console.WriteLine("\n***livre***");
    livre.AfficherDetails();
    Console.WriteLine("\n***livre2***");
    livre2.AfficherDetails();

    Console.ReadKey();
}
```

C#

Structure

- Par défaut, les membres d'une structure sont privés.
- Nous pouvons définir des constructeurs dans une structure, des getters, des setters, des constantes...

C#

Exemple de structures **C#** prédéfinies

- `Int32`
- `Boolean`
- `Double`
- `Char`
- ...