# C# : la programmation objet

**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

# Classe

## Particularité de C#

- La redéfinition et la surcharge sont possibles.
- La définition des attributs (appelés champs ici) 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.

# Classe

## Particularité de 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.

# Classe

## Déclaration

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

# Classe

## Déclaration

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

## Contenu d'une classe

- Champs (attributs)
- Constructeur(s) (et destructeur)
- Getters / Setters
- Autres méthodes

# Classe

**Exemple : classe `Personne` + attributs**

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

# Classe

**Exemple : classe `Personne` + constructeurs**

```
public class Personne
{
    // les attributs, ensuite
    public Personne()
    {
    }
    public Personne(string nom, string prenom)
    {
        this.nom = nom;
        this.prenom = prenom;
    }
    // on peut ajouter plusieurs autres
}
```

Le mot-clé `this` fait référence à l'objet courant.

# Classe

**Exemple : classe `Personne` + getters / setters**

```
public class Personne
{
    private int num;
    // autres attributs + constructeurs
    public int Num {
        get
        {
            return this.num;
        }
        set
        {
            this.num = value;
        }
    }
}
```

`value` est un mot clé permettant de récupérer la valeur envoyée par l'utilisateur

# Classe

**Le code précédent peut être factorisé**

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

Plus besoin de déclarer l'attribut `num`

# Classe

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

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

# Classe

## Pour créer une classe sous *Visual Studio Community 2017*

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

# Classe

## Pour générer un constructeur sous *Visual Studio Community 2017*

- 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
- Valider

# Classe

## Pour générer les getters/setters sous *Visual Studio Community 2017*

- 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`
- Valider en cliquant sur `Appliquer`

# Classe

## 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

# Classe

## Création d'objet à partir d'une classe : première méthode

```
// utilisation de constructeur sans paramètres
Personne p = new Personne();
// utilisation de setter
p.Nom = "wick";
p.Prenom = "john";
p.Num = 100;
// utilisation de constructeur avec deux parametres
Personne p2 = new Personne("bob", "mike");
// utilisation de getter
Console.WriteLine(p2.Nom); //affiche bob
```

# Classe

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

```
// les parenthèses du constructeur ne sont pas
   obligatoires
Personne p = new Personne()
{
    Nom = "wick",
    Prenom = "john",
    Num = 100
};
Console.Write($" je m'appelle { p.Prenom } { p.Nom }");
```

# Classe

## Classe abstraite

- Une classe peut être déclarée abstraite en utilisant le mot clé `abstract`.
  Par exemple : `abstract class Personne`.
- Une méthode peut être déclarée abstraite en ajoutant le mot clé `abstract`. Dans ce cas, la classe, elle aussi, doit obligatoirement être déclarée abstraite.

# Classe

## Classe partielle

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

# Classe

**Exemple : dans un fichier** `Class1.cs`

```
partial class Personne
{
    public string Prenom
    {
        get;set;
    }
}
```

**Exemple : dans un fichier** `Class2.cs`

```
partial class Personne
{
    public string Nom
    {
        get;set;
    }
}
```

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

# Classe

**Rien ne change pour l'instanciation** `Program.cs`

```
class Program
{
    static void Main(string[] args)
    {
        Personne personne = new Personne();
        personne.Nom = "White";
        personne.Prenom = "Carol";
        Console.WriteLine($"Bonjour {personne.
           Prenom} {personne.Nom}");
        Console.ReadKey();
    }
}
```

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

# Classe

**Si on crée un constructeur à deux paramètres dans** `Class1.cs`

```
partial class Personne
{
    public Personne(string prenom, string nom)
    {
        Prenom = prenom;
        Nom = nom;
    }
    public string Prenom
    {
        get;set;
    }
}
```

Le constructeur par défaut est écrasé, donc le code du `Main` sera signalé en rouge.

# Classe

## Méthode partielle

- Une méthode est dite partielle (déclarée avec le mot clé `partial`) si elle est définie dans une classe partielle, tandis que son implémentation est définie dans une autre classe partielle.

# Classe

**Dans le fichier** `Class1.cs`

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

**Dans le fichier** `Class2.cs`

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

# Classe

## 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.

# Classe

## Classe `sealed`

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

# Indexeur

## Définition

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

# Indexeur

```
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] = p;
mesAmis[1] = p2;
```

# Indexeur

**Code** `ListePersonnes`

```
class ListePersonnes
{
  private int nbrPersonnes;
  private Personne[] Personnes;

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

  public int NbrPersonnes
  {
    get => nbrPersonnes;
    set => nbrPersonnes = value;
```

# Indexeur

**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.

# Indexeur

## Pour tester

```
Personne p = new Personne();
p.Nom = "wick";
p.Prenom = "john";
p.Num = 100;
Personne p2 = new Personne(200,"bob", "mike");

ListePersonnes mesAmis = new ListePersonnes(2);
mesAmis[0] = p;
mesAmis[1] = p2;

for(int i=0; i < mesAmis.NbrPersonnes; i++)
{
    Console.WriteLine("Amis {0} est {1} {2}",i,
       mesAmis[i].Prenom, mesAmis[i].Nom);
}
```

# Interface

## 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)

# Interface

## Pour créer une interface sous *Visual Studio Community 2017*

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

# Interface

## Créer une interface

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

## Implémenter une interface

```
public class Personne : ISalutation
{
```

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

```
public void DireBonjour()
{
  Console.WriteLine("Bonjour {0} {1}", prenom, nom);
}
```

# Interface

## Créer une deuxième interface

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

## Implémenter plusieurs interfaces

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

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

```
void IGreeting.SayHello()
{
    Console.WriteLine("Hello {0} {1}", prenom, nom);
}
```

# Interface

## Appeler la méthode `DireBonjour()`

```
Personne p = new Personne();
p.Nom = "wick";
p.Prenom = "john";
p.Num = 100;
p.DireBonjour();
```

# Interface

## Appeler la méthode `DireBonjour()`

```
Personne p = new Personne();
p.Nom = "wick";
p.Prenom = "john";
p.Num = 100;
p.DireBonjour();
```

## Appeler la méthode `SayHello()`

```
IGreeting p2 = new Personne(200,"bob","mike");
p2.SayHello();
```

# Interface

## Appeler la méthode `DireBonjour()`

```
Personne p = new Personne();
p.Nom = "wick";
p.Prenom = "john";
p.Num = 100;
p.DireBonjour();
```

## Appeler la méthode `SayHello()`

```
IGreeting p2 = new Personne(200,"bob","mike");
p2.SayHello();
```

`p2` ne peut appeler `DireBonjour()`

# Interface

**On peut aussi faire**

```
Personne p2 = new Personne(200,"bob","mike");
((IGreeting)p2).SayHello();
```

# Interface

**Si par exemple `IGreeting` implémente une autre interface**

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

**L'interface `IMainMethod`**

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

# Interface

**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 personne doit implémenter aussi la méthode `SayAnything` de l'interface `IMainMethod`

# L'interface IEnumerable

**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()`

# L'interface IEnumerable

## Implémentons l'interface IEnumerable

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

# L'interface IEnumerable

## Implémentons l'interface IEnumerable

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

C'est signalé en rouge ?

# L'interface IEnumerable

**Implémentons l'interface IEnumerable**

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

C'est signalé en rouge ?

## Solution

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

# L'interface IEnumerable

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

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

# L'interface IEnumerable

**Maintenant, ce code est exécutable et n'est plus signalé 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)

# L'interface IEnumerable

**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;
}
```

**On appelle cette méthode**

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

et elle affiche 256

# L'interface `IEnumerable`

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

# L'interface `IEnumerable`

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

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

# L'interface IEnumerable

**Et si on veut 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`**

# L'interface IEnumerable

**Et si on veut 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**

# L'interface IEnumerable

**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;
    }
}
```

# L'interface IEnumerable

**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 i in  Power(2, 8))
{
    Console.Write("{0} ", i);
}
Console.ReadKey();
```

et elle affiche 2 4 8 16 32 64 128 256

# Interface

## Remarque

- une interface peut aussi être déclarée partielle.

# Héritage

## Héritage en C#

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

# Héritage

**Syntaxe : classe `Etudiant` qui hérite de la classe `Personne`**

```
public class Etudiant : Personne
{
    private int bourse;
```

**Définir un constructeur qui utilise le constructeur de la classe mère**

```
    public Etudiant(int num, string nom, string
       prenom, int bourse) : base(num,nom,prenom)
    {
        this.bourse = bourse;
    }
```

`base` : fait appel au constructeur de la classe mère

# Polymorphisme

## 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`

# Polymorphisme

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

```
public void DireBonjour()
{
    Console.WriteLine("Bonjour l'étudiant {0} {1}",
       Prenom, Nom);
}
```

# Polymorphisme

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

```
public void DireBonjour()
{
    Console.WriteLine("Bonjour l'étudiant {0} {1}",
       Prenom, Nom);
}
```

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

# Polymorphisme

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

```
public void DireBonjour()
{
    Console.WriteLine("Bonjour l'étudiant {0} {1}",
       Prenom, Nom);
}
```

La méthode est signalée en rouge 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 DireBonjour()
{
    Console.WriteLine("Bonjour l'étudiant {0} {1}",
       Prenom, Nom);
}
```

# Polymorphisme

## Faisons le test

```
Etudiant e1 = new Etudiant(100, "wick", "john", 450)
   ;
e1.DireBonjour();

// affiche Bonjour l'étudiant john wick

Personne e2 = new Etudiant(200,"bob", "mike",450);
e2.DireBonjour();

// affiche Bonjour mike bob
```

Et si on veut que la méthode `DireBonjour()` de la classe `Etudiant` soit exécutée pour `e2` (car il est aussi étudiant)

# Polymorphisme

**Modifions la méthode `DireBonjour()` dans `Etudiant`**

```
public override void DireBonjour()
{
    Console.WriteLine("Bonjour l'étudiant {0} {1}",
       Prenom, Nom);
}
```

**Modifions la méthode `DireBonjour()` dans `Personne`**

```
public virtual void DireBonjour()
{
    Console.WriteLine("Bonjour l'étudiant {0} {1}",
       Prenom, Nom);
}
```

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

# Polymorphisme

## Faisons le test

```
Etudiant e1 = new Etudiant(100, "wick", "john", 450)
   ;
e1.DireBonjour();

// affiche Bonjour l'étudiant john wick

Personne e2 = new Etudiant(200,"bob", "mike",450);
e2.DireBonjour();

// affiche Bonjour l'étudiant mike bob
```

# Structure

## 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

# Structure

## 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

# Structure

## Pour créer une structure sous *Visual Studio Community 2017*

- 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

# Structure

## Une première structure

```
using System;
namespace TestStruct
{
    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}");
        }
    }
}
```

# Structure

## Instancier une structure = instancier une classe

```
namespace TestStruct
{
    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();
        }
    }
}
```

# Structure

## 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`
- ...

# Structure

**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();
}
```

# Structure

## Structure

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