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 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.
- Le mot-clé `this` n'est pas obligatoire en l'absence d'ambiguïté.

C#

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.

C#

Déclaration

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

C#

Déclaration

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

Contenu d'une classe

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

C#

Pour créer une classe sous *Visual Studio Community 2019*

- 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.ts`

```
namespace MonProjet
{
    class Personne
    {
    }
}
```

C#

Ajoutons les trois attributs suivants à la classe `Personne`

```
namespace MonProjet
{
    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

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();
```

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

C#

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

```
namespace MonProjet
{
    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#

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

```
namespace MonProjet
{
    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#

Quelques conventions en programmation objets

- Attributs non-publiques
- Getters et setters publiques

C#

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

```
namespace MonProjet
{
    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 MonProjet
{
    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 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 (et utiliser la propriété)`
- Valider en cliquant sur `Appliquer`

Générons les getters et setters dans `Personne`

```
namespace MonProjet
{
    public 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; }
    }
}
```

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

Générons les getters et setters dans `Personne`

```
namespace MonProjet
{
    public 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; }
    }
}
```

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

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 supprimer les attributs et remplacer les getters et setters précédents par les suivants

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

C#

Si les getters et setters ne contiennent pas un traitement particulier, on peut supprimer les attributs et remplacer les getters et setters précédents par les suivants

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

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#

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 2019*

- 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

C#

Nouveau contenu de la classe `Personne`

```
namespace MonProjet
{
    public 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;
        }
    }
}
```

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

C#

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

```
namespace MonProjet
{
    public 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()
        {
        }
    }
}
```

C#

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

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

C#

Récapitulatif

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

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; }
```

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 MonProjet
{
    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($"Je m'appelle { personne.Prenom } {
              personne.Nom }");
            Personne personne2 = new Personne(200, "bob", "mike");
            Console.WriteLine(Personne.NbrPersonnes);
            Console.WriteLine($"Je m'appelle { personne2.Prenom } {
              personne2.Nom }");
            Console.ReadKey();
        }
    }
}
```

C#

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.

C#

Classe partielle

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

C#

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.

C#

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

C#

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.

C#

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.

C#

Dans le fichier `Class1.cs`

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

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#

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;
```

Contenu de la classe `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...