

L'orienté Objet en C#

Mme Ben bouzid NOURHENE



Plan:

- Les principes de la POO
- La notion de classes
- Constructeur, destructeur
- Garbage Collector
- Classes abstraites
- Interfaces
- Colle

Les principes de la POO:

Les principes fondamentaux de la programmation orientée objet sont les suivants :

- **Encapsulation** Masquer l'état interne et les fonctionnalités d'un objet et autoriser uniquement l'accès via un ensemble public de fonctions.
- **Héritage** Possibilité de créer de nouvelles abstractions basées sur des abstractions existantes.
Class A : Class B {.....}.
- **Polymorphisme** Possibilité d'implémenter des propriétés ou des méthodes héritées de différentes façons sur plusieurs abstractions.

La notion de classes:

- Les classes sont introduites avec le mot clé « class »
- Les données membres se déclarent comme des variables
- Plusieurs types de mode d'accès :
 - ❖ **private** (accessibles de la classe, par défaut)
 - ❖ **public** (accessibles de tout le monde),
 - ❖ **protected** (accessibles des classes dérivées),
 - ❖ **internal** (accessibles depuis l'assemblée).
- Les méthodes peuvent être surchargées
- Le mode de passage des paramètres doit être fourni à l'appel explicitement

La notion de classes:

➤ Syntaxe:

```
public classe NomClasse  
{ Methods, fields, properties and events.  
.....  
}
```

➤ Instanciation: keyword **new**

```
NomClasse inst = new NomClasse();
```

La notion de classes:

Program.cs

Program

```
using System;
```

4 références

```
public class Etudiant
```

```
{
```

5 références

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

5 références

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

6 références

```
public float Moyenne { get; set; }
```

1 référence

```
public Etudiant(string name, int age, float moyenne)
```

```
{
```

```
    Name = name;
```

```
    Age = age;
```

```
    Moyenne = moyenne;
```

```
}
```

```
// Other properties, methods,
```

```
}
```

0 références

```
class Program
```

```
{
```

0 références

```
static void Main()
```

```
{
```

```
    Etudiant etudiant1 = new Etudiant("Leopold", 6, 17.25f);
```

```
    Console.WriteLine("etudiant1 Name = {0} Age = {1} Moyenne = {2} ", etudiant1.Name, etudiant1.Age, etudiant1.Moyenne);
```

```
    // Declare new person, assign person1 to it.
```

```
    Etudiant etudiant2 = etudiant1;
```

```
    // Change the name of person2, and person1 also changes.
```

```
    etudiant2.Name = "Molly";
```

```
    etudiant2.Moyenne = 18.75f;
```

```
    etudiant1.Age = 16;
```

```
    etudiant1.Moyenne = 9;
```

```
    Console.WriteLine("Etudiant2 Name = {0} Age = {1} Moyenne = {2} ", etudiant2.Name, etudiant2.Age, etudiant2.Moyenne);
```

```
    Console.WriteLine("Etudiant1 Name = {0} Age = {1} Moyenne = {2} ", etudiant1.Name, etudiant1.Age, etudiant1.Moyenne);
```

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

```
}
```

```
}
```

file:///c:/users/nourhene ben bouzid/documents/visual studio 2013/Projects/Person_Classe/Person_Classe/bin/Debug/Person_Classe.EXE

```
etudiant1 Name = Leopold Age = 6 Moyenne = 17,25
```

```
Etudiant2 Name = Molly Age = 16 Moyenne = 9
```

```
Etudiant1 Name = Molly Age = 16 Moyenne = 9
```

Constructeur

- Une méthode spéciale d'une classe qui est automatiquement invoquée chaque fois qu'une instance de classe est créée.

Différence entre constructeur et méthode

Constructeur	Méthode
Un constructeur est utilisé pour initialiser un objet	Une méthode est utilisée pour exposer le comportement d'un objet
Le constructeur ne doit pas avoir un type de retour.	La méthode peut avoir un type de retour.
Un constructeur doit être identique au nom de la classe	Le nom de la méthode peut être identique ou non au nom de la classe
Un constructeur est invoqué implicitement.	Une méthode est appelée explicitement.

- Il est utilisé pour affecter des valeurs initiales aux données membres de la classe.

Constructeur

Voici la liste des constructeurs en C# :

- ❖ **Constructeur par défaut:** Un constructeur sans aucun paramètre. Si nous ne créons pas de constructeur, la classe appellera automatiquement le constructeur par défaut créer par le compilateur lorsqu'un objet est créé.
- ❖ **Constructeur paramétré:** Un constructeur avec au moins un paramètre.
- ❖ **Constructeur de copie:** Un constructeur qui crée un objet en copiant les variables d'un autre objet s'appelle un constructeur de copie.
- ❖ **Constructeur statique:** utilisé pour initialiser des données statiques.
- ❖ **Constructeur privé** utilisé dans les classes contenant uniquement des membres statiques. Si une classe a un ou plusieurs constructeurs privés et aucun constructeur public, les autres classes ne peuvent pas créer des instances de cette classe.

Destructeur

- Les destructeurs sont utilisés pour détruire les objets. Un destructeur est déclaré comme suit :

```
~MaClasse()  
{  
    // Instructions de destruction.  
}
```

Ils portent le même nom que la classe, mais sont précédés par **~**.

Garbage collector

Le Garbage collector (GC) est chargé de gérer la mémoire de notre application. Pour ce faire, il recherche les objets qui ne sont plus utilisés par l'application et les détruit. Il évite ainsi les fuites de mémoire.

- Le **Common Language Runtime** (CLR) .NET réserve une certaine quantité de mémoire qui sera utilisée par les objets de notre application. Si notre appli a besoin de plus de mémoire, le CLR préfère libérer de la mémoire qu'augmenter la quantité de mémoire nécessaire. C'est là qu'intervient le GC.
- Le GC utilise 3 segments de mémoire appelés génération:
 - ❖ **Génération 0**: identifie un objet nouvellement créé qui n'a jamais été marqué pour la collecte. Le GC s'exécute souvent pour ces objets.
 - ❖ **Génération 1**: identifie un objet qui a survécu à un GC
 - ❖ **Génération 2**: identifie les objets de grande tailles et à longue durée de vie

Garbage collector

- **GC.Collect()** permet de forcer l'exécution du GC.
- Lorsque notre code contient des objets utilisant des ressources non managées, la gestion de la mémoire est manuelle. Les ressources non managées sont par exemple des ressources systèmes, l'accès à des fichiers ou des connexions aux bases de données.

Solutions:

- implémenter la méthode Dispose de l'interface IDisposable. Cette méthode devra contenir le code chargé de libérer explicitement la mémoire des ressources non managées.
- créer un destructeur. Il sera appelé automatiquement lors de l'exécution du Garbage collector.

Classe Abstraite

- Une classe déclarée avec le mot-clé « **abstract** » s'appelle une classe abstraite en C#.
- Elle ne peut pas être instancié.
- Elle peut avoir des méthodes abstraites et non abstraites.
- L'utilisation des classe abstraite est au moment de l'héritage.
- Elle doit être hériter et ses méthode doivent êtres implémentés.
- On doit utiliser le mot-clé « **override** » avant la méthode qui est déclarée abstraite dans la classe fille.
- Elle peut contenir des constructeurs et des destructeurs.
- Elle ne peut pas être statique.

Classe Abstraite

- Les membres définis comme abstraits doivent être implémentés par des classes non abstraites dérivées de la classe abstraite.
- Il n'est pas possible de modifier une classe abstraite avec le modificateur sealed, car les deux modificateurs ont des significations opposées.
- utilisez le modificateur abstract pour indiquer que la méthode ou la propriété ne contient pas d'implémentation.
- Les déclarations de méthodes abstraites sont autorisées uniquement dans les classes abstraites.
- la déclaration d'une méthode se termine simplement par un point-virgule, et la signature n'est pas suivie d'accolades ({ })

Classe Abstraite

C#

Copy

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}

// Output: Area of the square = 144
```


Classe Abstraite

```
Program.cs [X]
Class_Abstrct_Eg.Car

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Class_Abstrct_Eg
{
    3 références
    public abstract class Vehicle {
        4 références
        public abstract void display();
    }

    1 référence
    public class Car : Vehicle {
        4 références
        public override void display() {
            Console.WriteLine("Vehicle : Car");
        }
    }
}
```

```
Program.cs [X]
Class_Abstrct_Eg.MyClass Main()

3 références
public abstract class Vehicle {
    4 références
    public abstract void display();
}

1 référence
public class Car : Vehicle {
    4 références
    public override void display() {
        Console.WriteLine("Vehicle : Car");
    }
}

1 référence
public class Train : Vehicle {
    4 références
    public override void display() {
        Console.WriteLine("Vehicle : Train");
    }
}

0 références
public class MyClass {
    0 références
    public static void Main() {
        Vehicle vh;
        vh = new Car();
        vh.display();
        vh = new Train();
        vh.display();
        Console.ReadKey();
    }
}
```

file:///c:/users/nourhene ben bouzid/documents/visual studio 2013/Projects/

Vehicle : Car
Vehicle : Train


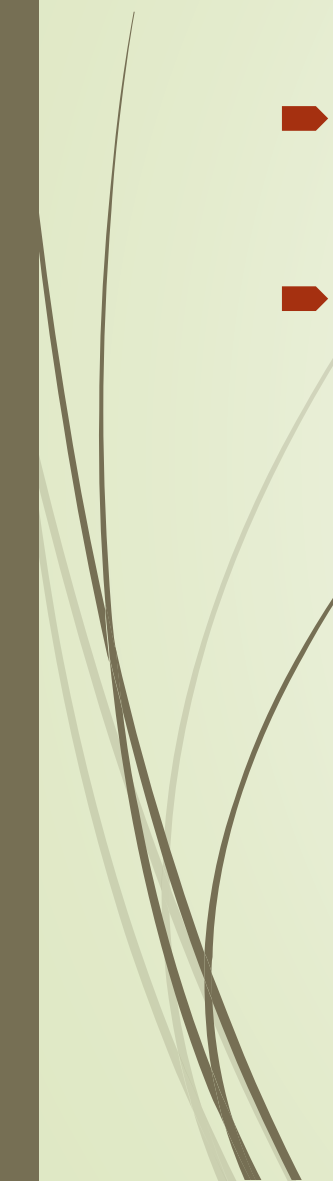
Interfaces

- Une interface définit un contrat. Une classe qui implémente une interface doit adhérer à son contrat.
- En C#, une interface peut être définie à l'aide du mot clé **interface**.

```
0 références
interface MyInterface
{
    /* Toutes les méthodes sont publiques par défaut
    * Et ils n'ont pas de corps
    */
    0 références
    void method1();
    0 références
    void method2();

    0 références
    void method3();
}
```

Interfaces

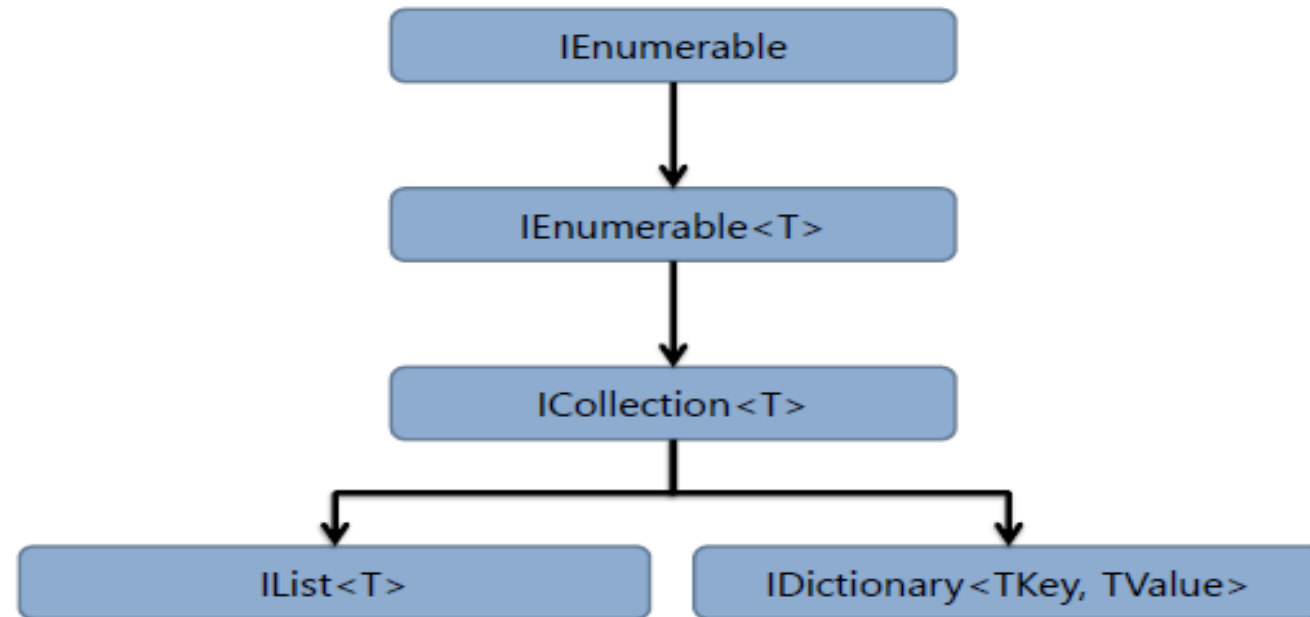
- 
- Une interface peut hériter de plusieurs interfaces de base, et une classe peut implémenter plusieurs interfaces.
 - Une classe qui implémente une interface doit implémenter toutes les méthodes de l'interface.
 - Une classe implémentant une interface qui hérite de plusieurs interfaces doit implémenter toutes les méthodes de l'interface et de ses interfaces parent
- 

Interface Polymorphism

- En ce qui concerne les interfaces, le polymorphisme indique que vous pouvez représenter une instance d'une classe comme une instance de n'importe quelle interface implémentée par la classe.
- Le polymorphisme d'interface peut aider à augmenter la flexibilité et la modularité de votre code.
- Supposons que vous ayez plusieurs classes qui implémentent une interface spécifique, vous pouvez écrire du code qui fonctionne avec n'importe laquelle de ces classes en tant qu'instances de l'interface sans connaître aucun détail de la classe implémenté.

Les collections IEnumerable, ICollection, IList and IDictionary

- Des Collections en C# qui ont tous des caractéristiques spécifiques qui les différencient et les rendent adaptables à certains scénarios.



Les collections IEnumerable, ICollection, IList and IDictionary

- **IEnumerable:** un conteneur qui peut contenir certains éléments. Dans IEnumerable, vous pouvez parcourir chaque élément et vous ne pouvez pas modifier les données. C'est le type de conteneur de liste le plus basique. Un IEnumerable ne contient même pas le nombre d'éléments dans la liste, à la place, vous devez itérer sur les éléments pour obtenir le nombre d'éléments.
- **ICollection:** Un autre type de collection, qui dérive de IEnumerable et étend ses fonctionnalités pour modifier des données. ICollection contient également le nombre d'éléments.

Les collections IEnumerable, ICollection, IList and IDictionary

- **IList:** Un IList peut effectuer toutes les opérations combinées de IEnumerable et ICollection, et quelques autres opérations comme l'insertion ou la suppression d'un élément au milieu d'une liste.
- **IDictionary:** Représente une collection générique de paires clé/valeur.



Merci pour votre attention