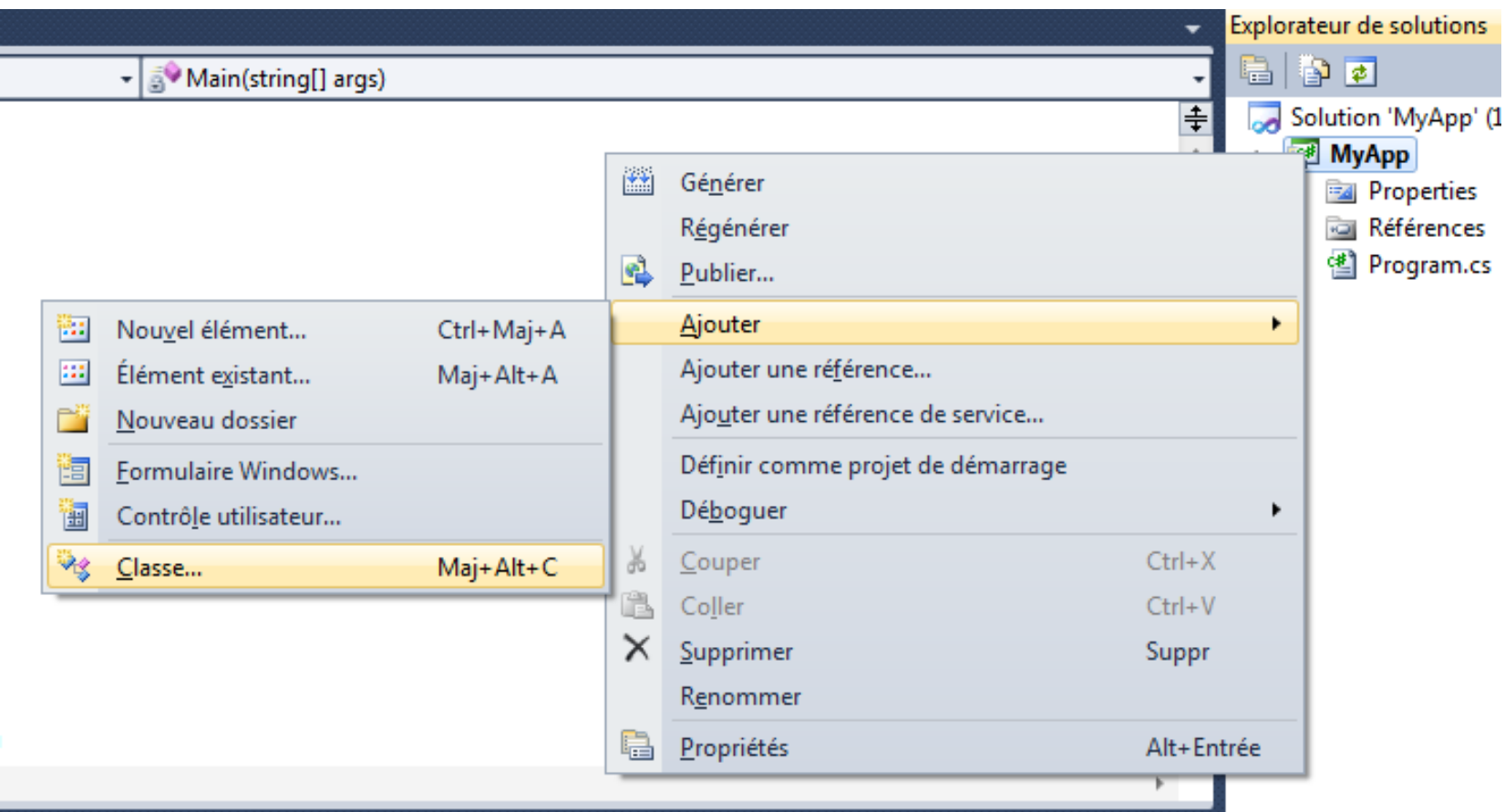


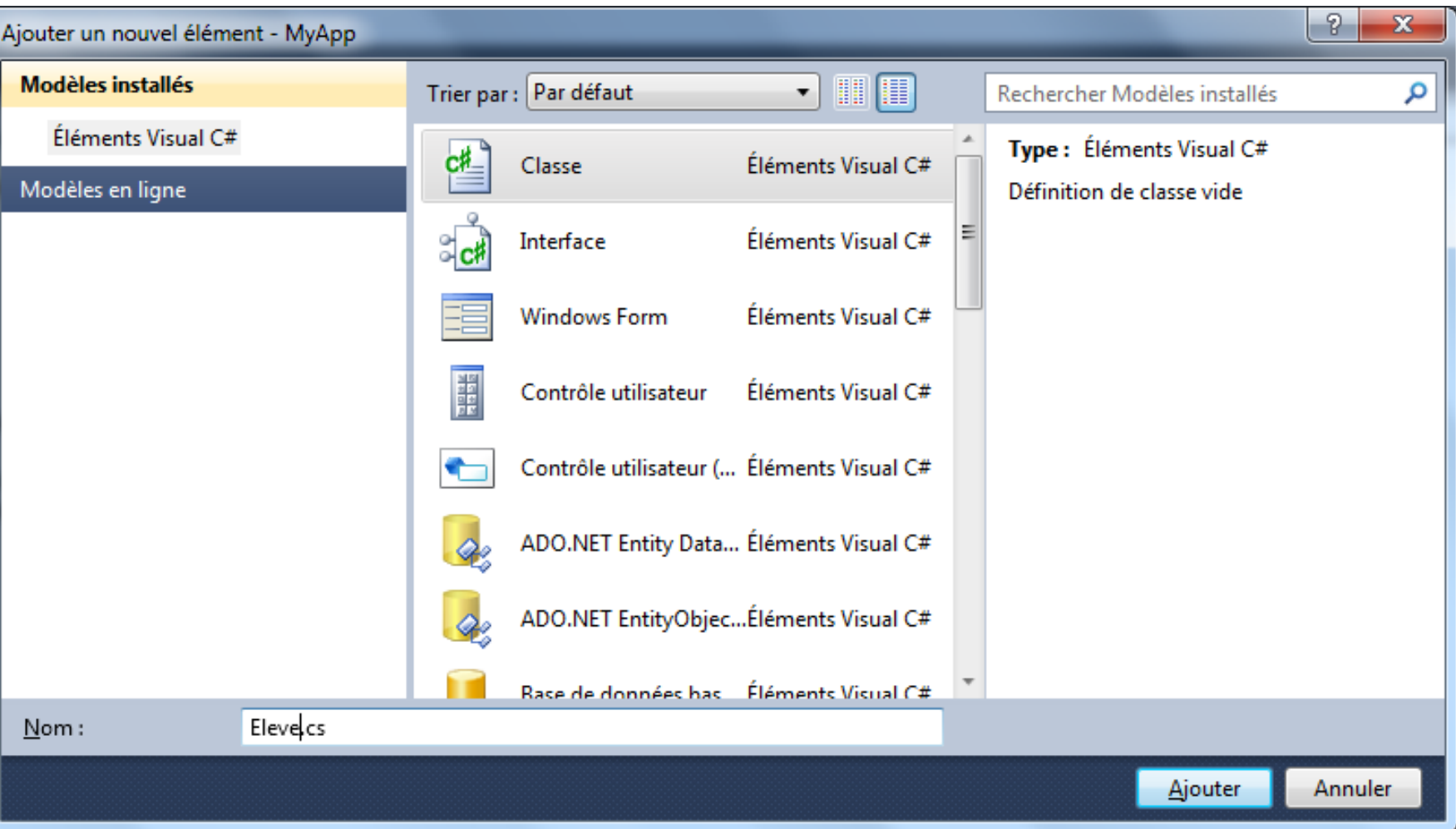
POO en C#

- Tout en C# est un objet d'une certaine classe
- Pour créer une classe :



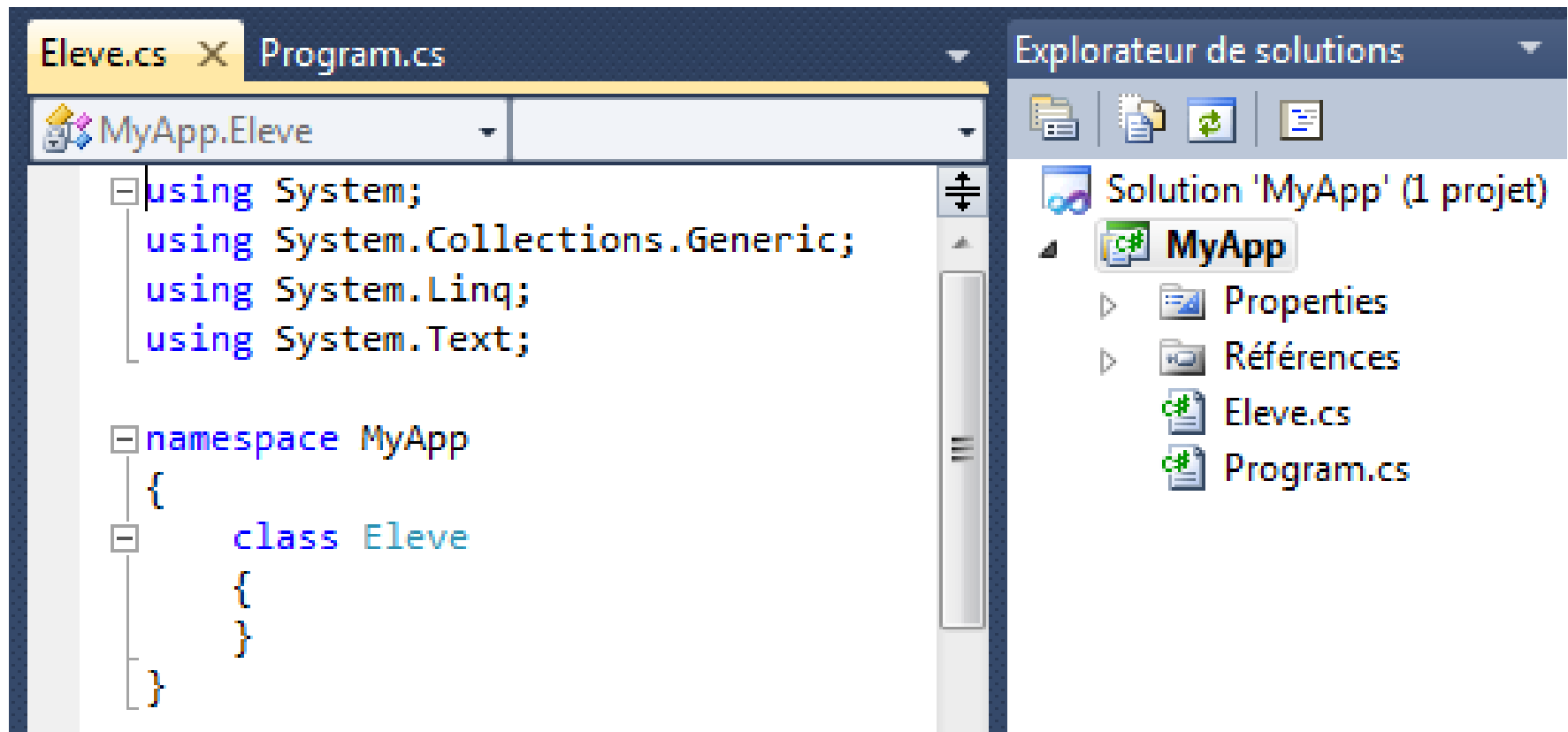
POO en C#

- Choisir le nom de la classe (Eleve)



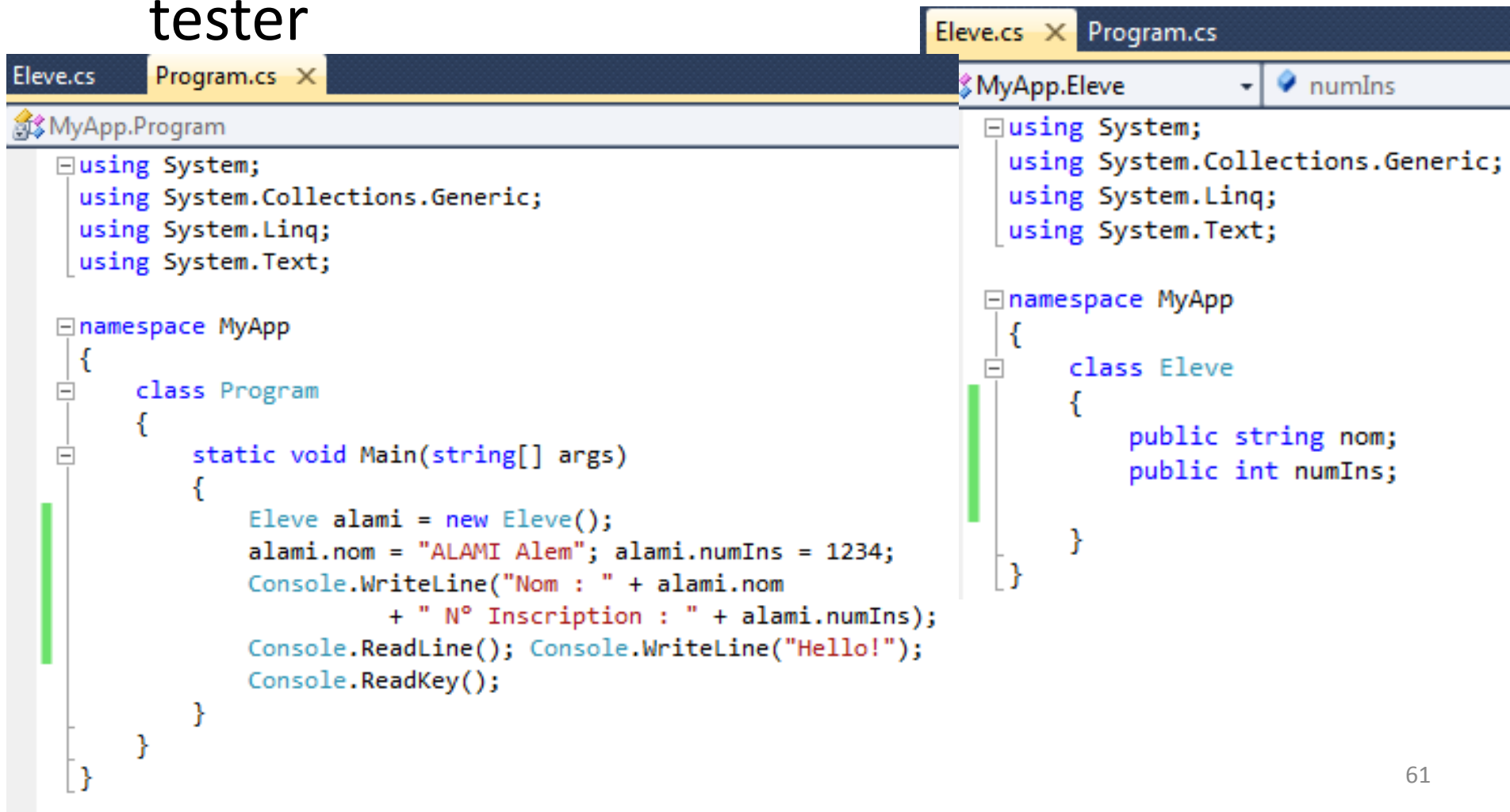
POO en C#

- Et la classe est créée



POO en C#

- Modifier les source Program.cs et Eleve.cs puis tester



```
Eleve.cs Program.cs X
MyApp.Eleve numIns
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MyApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Eleve alami = new Eleve();
            alami.nom = "ALAMI Alem"; alami.numIns = 1234;
            Console.WriteLine("Nom : " + alami.nom
                + " N° Inscription : " + alami.numIns);
            Console.ReadLine(); Console.WriteLine("Hello!");
            Console.ReadKey();
        }
    }
}

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

namespace MyApp
{
    class Eleve
    {
        public string nom;
        public int numIns;
    }
}
```

POO en C#








Notion de visibilité

Visibilité	Description
public	Accès non restreint
protected	Accès depuis la même classe ou depuis une classe dérivée
private	Accès uniquement depuis la même classe
internal	Accès restreint à la même assembly
protected internal	Accès restreint à la même assembly ou depuis une classe dérivée

POO en C#

- mettre en mode private les attributs
provoque des erreurs.

```
namespace MyApp
{
    class Eleve
    {
        private string nom;
        private int numIns;
    }
}
```

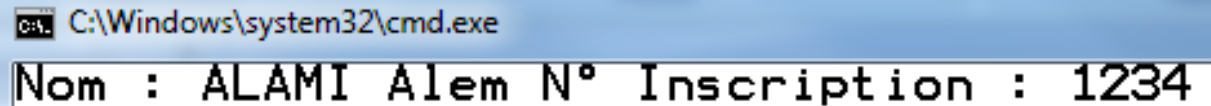
Liste d'erreurs	
 4 erreurs  0 avertissements  0 messages	
	Description
 1	'MyApp.Eleve.nom' est inaccessible en raison de son niveau de protection
 2	'MyApp.Eleve.numIns' est inaccessible en raison de son niveau de protection
 3	'MyApp.Eleve.nom' est inaccessible en raison de son niveau de protection
 4	'MyApp.Eleve.numIns' est inaccessible en raison de son niveau de protection

POO en C#

- Utiliser les propriétés :
 - ressemblent à des méthodes
 - dotés de get & set
 - value : la valeur disponible

```
namespace MyApp
{
    class Eleve
    {
        private string name;
        public string nom
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }

        private int nins;
        public int numIns
        {
            get
            {
                return nins;
            }
            set
            {
                nins = value;
            }
        }
    }
}
```



C:\Windows\system32\cmd.exe
Nom : ALAMI Alem N° Inscription : 1234

- Pour interdire la modification directe il suffit d'omettre le bloc set.

POO en C#

- Les propriétés auto-implémentées

```
private string nom;  
public string Nom  
{  
    get  
    {  
        return nom;  
    }  
    set  
    {  
        nom = value;  
    }  
}
```



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

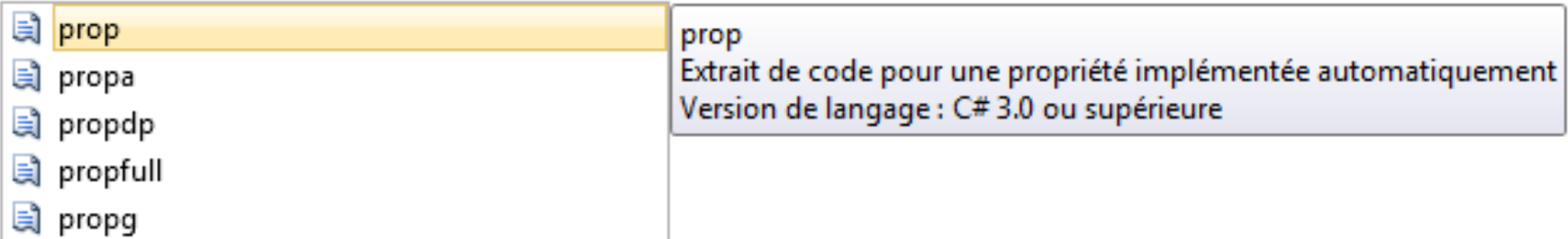
- Le compilateur se charge de générer le code nécessaire

POO en C#

- snippets : extraits de code. Taper **prop**

```
class Eleve
{
    public string Nom { get; set; }

    prop
    {
        prop
        propa
        propdp
        propfull
        propg
    }
}
```



- Enter -> Tab

```
class Eleve
{
    public string Nom { get; set; }

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

- Tester **propfull** et **propg**

POO en C#

- Syntaxe abrégée

```
class Eleve
{
    public string Nom { get; set; }
    public int numIns { get; set; }
}
```

```
Eleve alami = new Eleve();
alami.Nom = "ALAMI Alem";
alami.numIns = 1234;
```

```
Eleve alami = new Eleve { |
```

Nom	string Eleve.Nom
numIns	

```
Eleve alami = new Eleve { Nom = "ALAMI Alem", |
```

numIns	int Eleve.numIns
--------	------------------

↓

```
Eleve alami = new Eleve { Nom = "ALAMI Alem", numIns = 1234 };
```

POO en C#

Le constructeur

- Le constructeur est une méthode particulière qui permet d'initialiser les attributs d'un objet au moment de sa création.
 - Le constructeur a le même nom que la classe et qui ne possède pas de type de retour.
 - il est appelée lors de la création de l'objet, avec **new**.
- Le préfixe **this.** représente l'objet en cours de la classe.
 - Il permet de clarifier éventuellement le code et éviter une ambiguïté, mais Il est généralement facultatif.

POO en C#

Le constructeur

- Exp

```
class Eleve
{
    public string nom { get; set; }
    public int numIns { get; set; }

    public Eleve()
    {
        nom = "Vide";
        numIns = 0;
    }

    public Eleve(string nom, int num)
    {
        this.nom = nom;
        numIns = num;
    }
}
```

```
Eleve ev = new Eleve();
Console.WriteLine("Nom :" + ev.nom);
Console.WriteLine("N° :" + ev.numIns);

Eleve e = new Eleve("test",10);
Console.WriteLine("Nom :" + e.nom);
Console.WriteLine("N° :" + e.numIns);
```

POO en C#

Héritage : On utilise les ':'. Exemple

```
public class Animal
{
    public int NombreDePattes { get; set; }
    public void Manger()
    {
        Console.WriteLine("Je mange");
    }
}

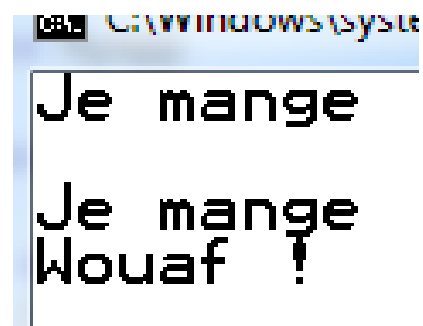
public class Chien : Animal
{
    public void Aboier()
    {
        Console.WriteLine("Wouaf !");
    }
}
```

POO en C#

Héritage : test

```
Animal animal = new Animal { NombreDePattes = 4 };  
animal.Manger();  
Console.WriteLine();  
Chien chien = new Chien { NombreDePattes = 4 };  
chien.Manger();  
chien.Aboyer();  
Console.ReadKey();
```

la classe Chien héritera de tout
ce qui est **public** ou **Protected**.



C:\windows\system32\cmd.exe
Je mange
Je mange
Wouaf !

POO en C#

Super objet : Object

- Tous les objets créés ou disponibles dans le framework .NET héritent d'un objet de base.
 - On parle en général d'un « super objet ».
- L'intérêt de dériver d'un tel objet est de permettre à tous les objets d'avoir certains comportements en commun, mais également de pouvoir éventuellement tous les traiter en tant qu'objet.
- Notre super objet est représenté par la classe Object .

POO en C#

Super objet : Object

```
public class ObjetVide  
{  
}
```

```
ObjetVide ov = new ObjetVide();
```

ov.

- Equals
- GetHashCode
- GetType
- ToString

```
string object.ToString()  
Retourne un System.String qui représente le System.Object actif.
```

Les méthodes proposées ont été hérité Object

POO en C#

Polymorphisme : substitution

- On veut redéfinir la méthode ToString()
 - substituer la méthode existante afin de remplacer son comportement → **override**

```
class Eleve
{
    public string Nom { get; set; }
    public int numIns { get; set; }

    public override string ToString()
    {
        return "Je suis un élève et je m'appelle " + Nom;
    }
}
```

POO en C#

Polymorphisme : substitution

```
static void Main(string[] args)
{
    Eleve alami = new Eleve { Nom = "ALAMI Alem", numIns = 1234 };
    Console.WriteLine(alami.ToString());
}
```



C:\Windows\system32\cmd.exe

Je suis un élève et je m'appelle ALAMI Alem

On peut écrire directement `Console.WriteLine(alami);`



C:\Windows\system32\cmd.exe

Je suis un élève et je m'appelle ALAMI Alem

POO en C#

Polymorphisme : substitution

- En réalité, il faut que la méthode à remplacer s'annonce comme candidate à la substitution.
 - on ne peut substituer n'importe quelle méthode
 - Il faut marquer la méthode par **virtual**
- Pour ToString, les concepteurs du framework .NET ont autorisé cette éventualité.

POO en C#

Polymorphisme : substitution

- Exp : classe Animal

```
public class Animal
{
    public int NombreDePattes { get; set; }

    public virtual void Manger()
    {
        Console.WriteLine("Mettre les aliments dans la bouche");
        Console.WriteLine("Mastiquer");
        Console.WriteLine("Avaler");
        Console.WriteLine("...");
    }
}
```

POO en C#

Polymorphisme : substitution

- Exp : classe Chien

```
public class Chien : Animal
{
    public void Aboier()
    {
        Console.WriteLine("Wouaf !");
    }

    public override void Manger()
    {
        Console.WriteLine("Réclamer à manger au maître");
        base.Manger();
        Console.WriteLine("Remuer la queue");
    }
}
```

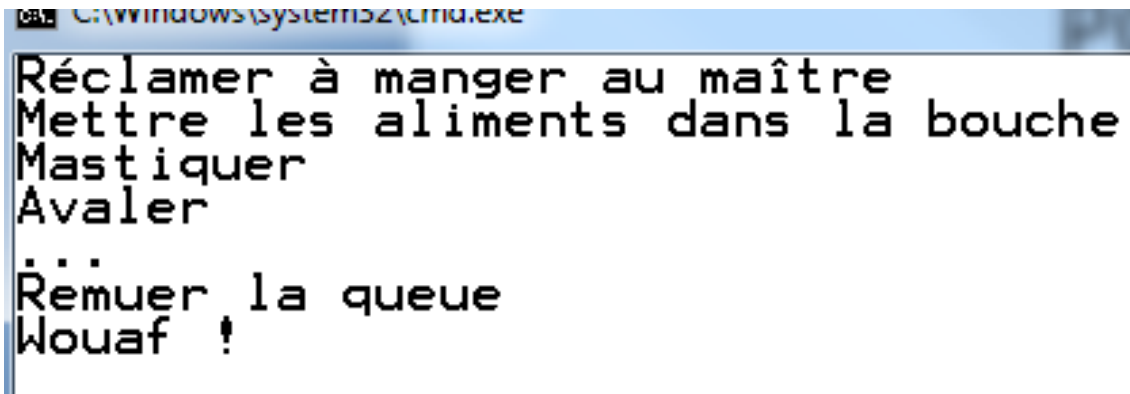
POO en C#

Polymorphisme : substitution

- Exp : classe Chien
 - le mot clé **base** fait référence à la classe mère
 - ainsi `base.Manger();` appelle la méthode `Manger()` de la classe mère afin de réutiliser son fonctionnement.

```
Chien chien = new Chien { NombreDePattes = 4 };  
chien.Manger();
```

```
chien.Aboyer();  
Console.ReadKey();
```



```
C:\windows\system32\cmd.exe  
Réclamer à manger au maître  
Mettre les aliments dans la bouche  
Mastiquer  
Avaler  
Remuer la queue  
Wouaf !
```

POO en C#

Constructeur et Base

- Un constructeur appeler le constructeur d'une classe de mère à l'aide de **base**.

- Par exemple :

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
```

- Dans une classe dérivée, si un constructeur de classe de base n'est pas appelé explicitement à l'aide du mot clé **base**, le constructeur par défaut, s'il existe, est appelé implicitement.

```
public Manager(int initialdata)
{
```

<=>

```
public Manager(int initialdata)
    : base()
{
```

POO en C#

Polymorphisme : surcharge

- possibilité de définir la même méthode avec des paramètres en entrée différents.
 - Méthode polymorphe (multi-forme)
- Exp : WriteLine possède 19 formes

`Console.WriteLine(`

▲ 1 sur 19 ▼ `void Console.WriteLine()`

Écrit le terminateur de la ligne active dans le flux de sortie standard.

POO en C#

Polymorphisme : surcharge

- Exp :

```
public class Math
{
    public int Addition(int a, int b)
    {
        return a + b;
    }
    public double Addition (double a, double b)
    {
        return a + b;
    }
}
```

POO en C#

Polymorphisme : surcharge

- Exp :

```
Math math = new Math();  
int a = 5;  
int b = 6;  
int resultat = math.Addition(a, b);  
double c = 1.5;  
double d = 5.0;  
double resultatDouble = math.Addition(c, d);
```

math.Addition(

▲ 1 sur 2 ▼ double Math.Addition(**double a**, double b)

POO en C#

Conversion entre objets :

- Un chien est un animal
- Le cast est possible

```
Chien bouby = new Chien();  
Animal animal = (Animal)bouby;
```

- On peut tester la nature d'un objet avec **is**

```
if (animal is Chien)  
{  
    Chien c = (Chien)animal;  
    c.Aboyer();  
}
```

POO en C#

Conversion entre objets :

- le cast dynamique : se fait en employant **as**
- Il vérifie que l'objet est bien convertible.
 - Si c'est le cas, alors il fait un cast explicite pour renvoyer le résultat de la conversion,
 - sinon, il renvoie une référence nulle.

```
Chien c = animal as Chien;  
if (c != null)  
{  
    c.Aboyer();  
}
```

POO en C#

Comparer des objets

- la comparaison en utilisant l'opérateur d'égalité `==` permet simplement de vérifier si les références pointent vers le même objet.
- La comparaison d'égalité entre deux objets, c'est en fait le rôle de la méthode `Equals()` dont chaque objet hérite de la classe mère `Object`.
- Le comportement de `Equals()` est à redéfinir.
 - Par défaut n'est rien d'autre qu'une comparaison de références

POO en C#

Comparer des objets

- Exp :

```
class Eleve
{
    public string Nom { get; set; }
    public int numIns { get; set; }

    public override bool Equals(object obj)
    {
        Eleve e = obj as Eleve;

        if (e == null)
            return false;

        return Nom == e.Nom && numIns == e.numIns;
    }
}
```

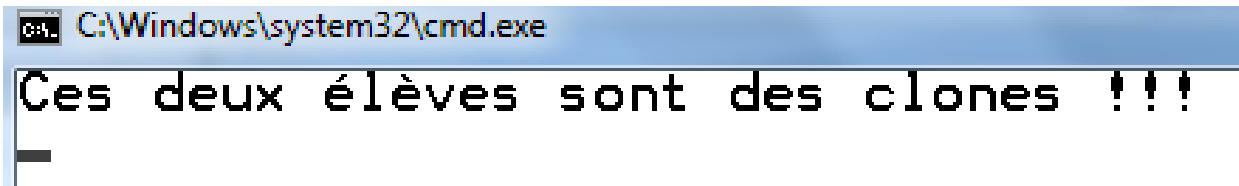
POO en C#

Comparer des objets

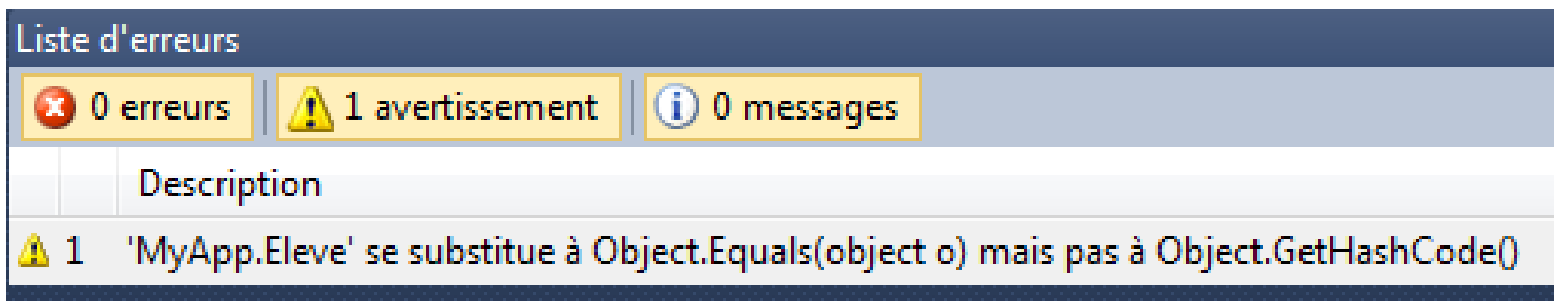
- Exp :

```
Eleve alami = new Eleve { Nom = "ALAMI Alem", numIns = 1234 };
Eleve alem = new Eleve { Nom = "ALAMI Alem", numIns = 1234 };

if (alami.Equals(alem))
    Console.WriteLine("Ces deux élèves sont des clones !!!");
```



A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The command prompt displays the text "Ces deux élèves sont des clones !!!" on a single line.



A screenshot of the Visual Studio error list window. The title bar says "Liste d'erreurs". Below the title bar, there are three summary boxes: "0 erreurs" (with a red X icon), "1 avertissement" (with a yellow warning triangle icon), and "0 messages" (with a blue information icon). Below these is a table with the following content:

	Description
1	'MyApp.Eleve' se substitue à Object.Equals(object o) mais pas à Object.GetHashCode()

POO en C#

Comparer des objets

- Le GetHashCode permet d'identifier un objet
 - Deux objets qui sont égaux retournent des codes de hachage égaux.
- Pour enlever l'avertissement, on se servira des GetHashCode des propriétés et on ajoutera :

```
public override int GetHashCode()  
{  
    return Nom.GetHashCode() * numIns.GetHashCode();  
}
```


POO en C#

Les interfaces

- Une interface est un contrat que s'engage à respecter un objet.
- Être comparable est un exemple de contrat
 - L'interface **IComparable** permet de définir un contrat de méthodes destinées à la prise en charge de la comparaison entre deux instances d'une classe.
- Pour s'assurer que nos objets seront correctement comparables, il faut que notre classe **implémente** cette interface.

```
class Eleve : IComparable  
{
```

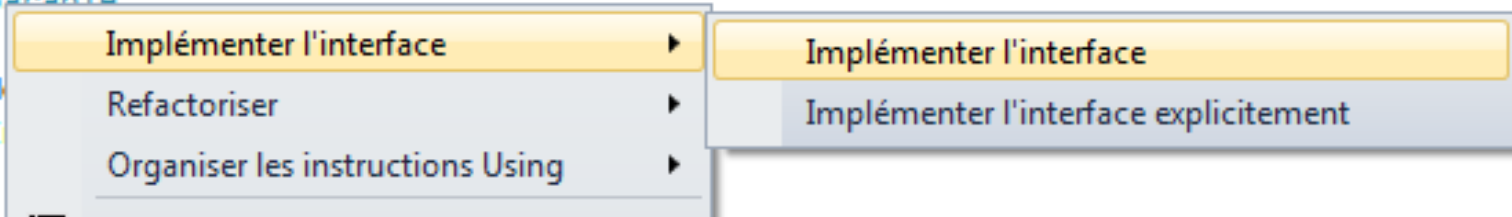
POO en C#

Les interfaces

- Notre classe ne contient que les propriétés.
- Faire clic droit sur l'interface

```
class Eleve : IComparable
{
    public string Nom { get; set; }
    public int numIns { get; set; }

    public int CompareTo(object obj)
    {
        throw new NotImplementedException();
    }
}
```



```
class Eleve : IComparable
{
    public string Nom { get; set; }
    public int numIns { get; set; }

    public int CompareTo(object obj)
    {
        throw new NotImplementedException();
    }
}
```

POO en C#

Les interfaces

- La méthode `CompareTo` doit retourner un valeur:
 - < 0 (-1) si un objet est inférieure à l'autre
 - $= 0$ s'il sont égaux
 - > 0 (1) si un objet est supérieure à l'autre

Critère de tri

```
class Eleve : IComparable
{
    public string Nom { get; set; }
    public int numIns { get; set; }

    public int CompareTo(object obj)
    {
        Eleve e = (Eleve) obj;
        if (numIns < e.numIns)
            return -1;
        if (numIns > e.numIns)
            return 1;
        return 0;
    }
}
```

POO en C#

Les interfaces

- Notre méthode compare une propriété entière
 - On peut simplifier notre code

```
public int CompareTo(object obj)
{
    Eleve e = (Eleve) obj;
    return numIns.CompareTo(e.numIns);
}
```

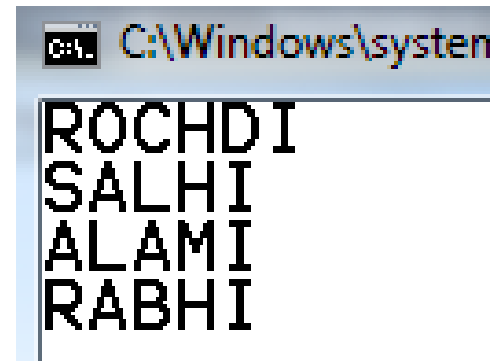
POO en C#

Les interfaces

- Test :

```
Eleve[] groupe = new Eleve[] { new Eleve { Nom = "ALAMI", numIns = 1234 },  
                                new Eleve { Nom = "RABHI", numIns = 2011 },  
                                new Eleve { Nom = "ROCHDI", numIns = 203 },  
                                new Eleve { Nom = "SALHI", numIns = 1101 }  
                                };
```

```
Array.Sort(groupe);  
foreach (Eleve e in groupe)  
{  
    Console.WriteLine(e.Nom);  
}
```



```
C:\Windows\system  
ROCHDI  
SALHI  
ALAMI  
RABHI
```

POO en C#

Les interfaces

- Une interface se définit avec **interface**
- Par convention le nom de l'interface commence

par I

```
public interface Ivehicule
{
    int nombreRoues { get; set; }
    void Rouler();
}
```

- On ne définit ni corps de méthode ni visibilité de propriétés

POO en C#

Les interfaces

- Exemple de classes implémentant l'interface

```
public class auto : Ivehicule
{
    public int nombreRoues { get; set; }
    public void Rouler()
    {
        Console.WriteLine("Je roule sur " + nombreRoues + "roues.");
    }
}
```

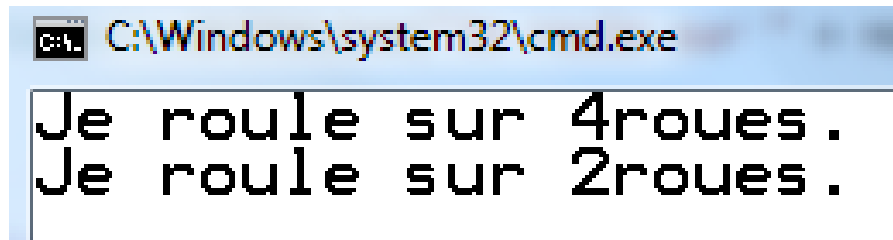
```
public class moto : Ivehicule
{
    public int nombreRoues { get; set; }
    public void Rouler()
    {
        Console.WriteLine("Je roule sur " + nombreRoues + "roues.");
    }
}
```

POO en C#

Les interfaces

- Exemple de classes implémentant l'interface

```
auto a = new auto {nombreRoues = 4 };  
moto m = new moto { nombreRoues = 2 };  
  
List<IVehicule> vh = new List<IVehicule> { a, m };  
  
foreach (IVehicule v in vh)  
{  
    v.Rouler();  
}
```



```
C:\Windows\system32\cmd.exe  
Je roule sur 4roues.  
Je roule sur 2roues.
```


POO en C#

Les interfaces

- Une interface peut hériter d'une autre ou plusieurs

```
public interface IAutoMobile : IVehicule
{
    |
}
```

- Une classe peut implémenter plusieurs interfaces mais ne peut hériter que d'une seule classe

```
public class Avion : IVehicule, IPlane
{
}
```

POO en C#

Classes et méthodes abstraites

- Une classe abstraite ne peut être instanciée car incomplète
 - Il lui manque du code
- Justement une méthode abstraite est une méthode qui ne contient pas d'implémentation
- Pour être utilisables, les classes abstraites doivent être héritées et les méthodes redéfinies.

POO en C#

Classes et méthodes abstraites

- En général, les classes abstraites sont utilisées comme classe de base pour d'autres classes.
- Si une classe possède une méthode abstraite, alors la classe doit absolument être abstraite
- Exp :

```
public abstract class Animal
{
    public abstract void SeDeplacer();
}
```

POO en C#

Classes et méthodes abstraites

- Exp : `public class Chat : Animal`

Refactoriser

Organiser les instructions Using

Implémenter une classe abstraite

```
public class Chat : Animal
{
    public override void SeDeplacer()
    {
        Console.WriteLine("je me déplace su 4 pattes.");
    }
}
```