Les bases des classes en C#

Programmation C#

Déclaration d'une classe

Espace de noms – Généralement associé au projet, mais pas seulement.
Peut contenir plusieurs classes.

Implicitement toutes les classes sont héritières de **Object**

```
using System;

namespace ClassesLesBases
{
    /// <summary>
    /// Description of CVecteur.
    /// </summary>
    public class CVecteur
    {
        //taille du tableau
        public int n;

        //tableau interne
        public double [] tab;

        //constructeur
        public CVecteur()
        {
        }
    }
}
```

Champs

Qualificateur de la classe :
public visible par n'importe quel programme d'un autres espace de noms
protected , pour les classes imbriquées, sera visible uniquement par la classe conteneur et ses héritiers
internal visible par les autres classes du même assembly (l'EXE ou la DLL du projet)
private visible uniquement par les classes du même espace de noms

Méthode : Un constructeur par défaut (sans paramètres) est toujours proposé

Instanciation

On est dans le même espace de noms

Déclaration de la variable + initialisation

Hum ! Danger …

```
using System;

namespace ClassesLesBases
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            CVecteur vec = new CVecteur();
            vec.n = 5;
            vec.tab = new Double[5];

            vec.tab[0] = 3.2;

            Console.WriteLine(vec.tab[0]);


            Console.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
        }
    }
}
```

Constructeur

Méthode appelée automatiquement lors de l'instanciation

```
namespace ClassesLesBases
{
    /// <summary>
    /// Description of CVecteur.
    /// </summary>
    public class CVecteur
    {
        //taille du tableau
        public int n;

        //tableau interne
        public double [] tab;

        //méthode de création du tableau
        private void initialisation(int taille){
            this.tab = new Double[taille];
        }

        //constructeur par défaut - vecteur de taille 5
        public CVecteur()
        {
            this.n = 5;
            this.initialisation(this.n);
        }

        //autre constructeur avec la taille
        public CVecteur(int taille){
            this.n = taille;
            this.initialisation(taille);
        }

        //constructeur avec taille et valeur d'initialisation
        public CVecteur(int taille, double valeur):this(taille){
            for(int i = 0; i < this.n; i++)
                this.tab[i] = valeur;
        }
    }
}
```

- Même nom que la classe.
- Forcément public
- Peuvent être plusieurs
- Se différencient par les paramètres
- On peut réutiliser un des constructeurs dans un autre constructeur

Réutilisation d'un autre constructeur

Destructeur

Méthode appelée automatiquement lors de la destruction

```
public class CVecteur
{
    //taille du tableau
    public int n;

    //tableau interne
    public double [] tab;

    //méthode de création du tableau
    private void initialisation(int taille)...

    //constructeur par défaut - vecteur de taille 5
    public CVecteur()...

    //autre constructeur avec la taille
    public CVecteur(int taille)...

    //constructeur avec taille et valeur d'initialisation
    public CVecteur(int taille, double valeur):this(taille)...

    //destructeur
    ~CVecteur(){
        this.tab = null;
        this.n = 0;

        Console.WriteLine("je détruis...");
    }
}
```

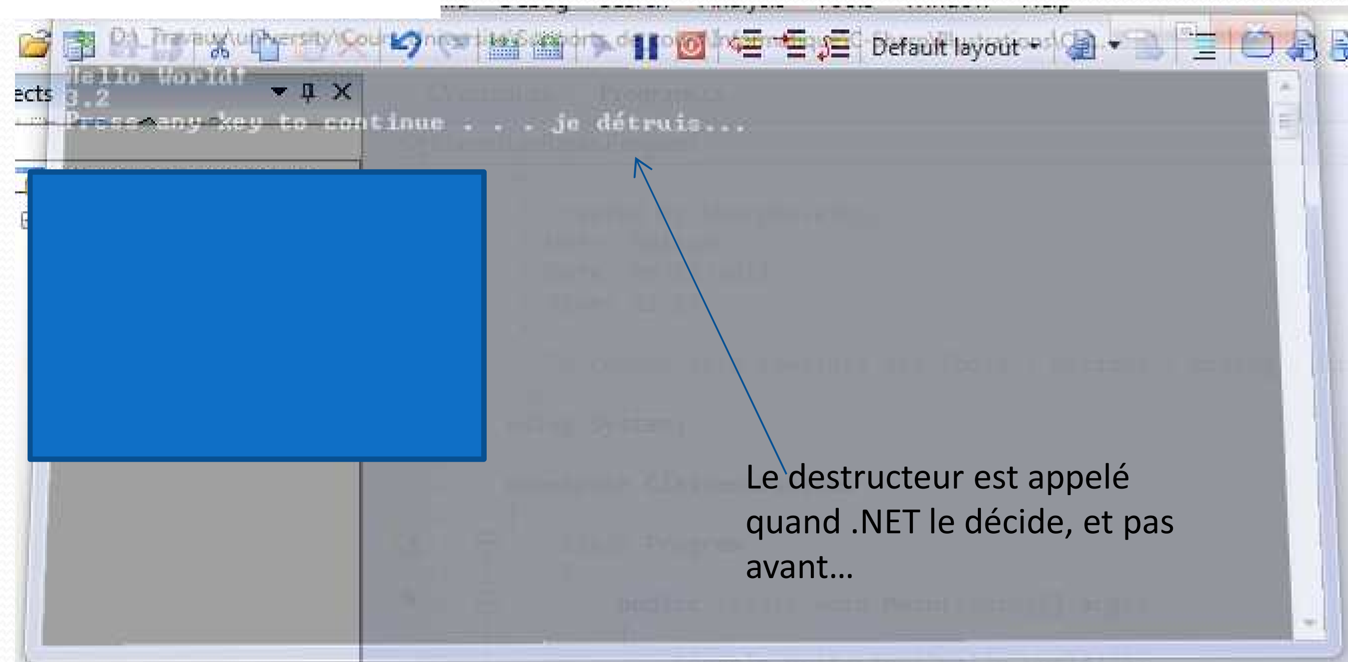
- Un seul destructeur par classe
- Pas de paramètres
- Jamais d'appel explicite

- Rarement nécessaire sauf nécessités particulières. Ex. pour fermer les fichiers temporaires, fermer une connexion, etc.

Destructeur

Fonctionnement

```csharp
namespace ClassesLesBases
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            CVecteur vec = new CVecteur(5);
            vec.tab[0] = 3.2;

            Console.WriteLine(vec.tab[0]);

            vec = null;

            Console.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
        }
    }
}
```



Le destructeur est appelé quand .NET le décide, et pas avant…

Encapsulation

Définir une portée pour les champs et les méthodes

- private
- protected
- public

```
public class CVecteur
{
    //taille du tableau
    private int n;
    //tableau interne
    private double [] tab;
    //méthode de création du tableau
    private void initialisation(int taille)...
    //constructeur par défaut - vecteur de taille 5
    public CVecteur()...
    //autre constructeur avec la taille
    public CVecteur(int taille)...
    //constructeur avec taille et valeur d'initialisation
    public CVecteur(int taille, double valeur):this(taille)...
    //destructeur
    ~CVecteur()...
    //lire les valeur
    protected double getValue(int i){
        if (i >= 0 && i < this.n)
            return this.tab[i];
        else
            return Double.NaN;
    }
    //inscrire une valeur
    protected void setValue(int i, double valeur){
        if (i >= 0 && i < this.n)
            this.tab[i] = valeur;
    }

}
```

Les champs devraient toujours être *private* (au pire *protected*)

Accesseurs

Encapsulation – Aller plus loin avec les propriétés

```
public class CVecteur
{
    //taille du tableau
    private int n;
    //tableau interne
    private double [] tab;
    //méthode de création du tableau
    private void initialisation(int taille)...
    //constructeur par défaut - vecteur de taille 5
    public CVecteur()...
    //autre constructeur avec la taille
    public CVecteur(int taille)...
    //constructeur avec taille et valeur d'initialisation
    public CVecteur(int taille, double valeur):this(taille)...
    //destructeur
    ~CVecteur()...
    //lire les valeur
    protected double getValue(int i)...
    //inscrire une valeur
    protected void setValue(int i, double valeur)...

    //propriété taille du vecteur
    public int size{
        get {
            return this.n;
        }
        set{
            if (value > 0){
                Array.Resize(ref this.tab,value);
                this.n = value;
            }
        }
    }

    //accès aux valeurs
    public double this[int i]{
        get{
            return this.getValue(i);
        }
        set{
            this.setValue(i,value);
        }
    }
}
```

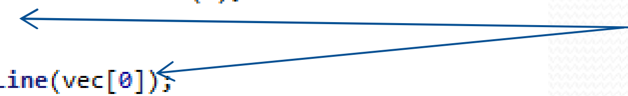
value est un mot clé

Pour un champ atomique

Pour un champ vecteur : **indexeur**

Encapsulation – Aller plus loin avec les propriétés (2)

```
namespace ClassesLesBases
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            CVecteur vec = new CVecteur(5);
            vec[0] = 3.2;

            Console.WriteLine(vec[0]);

            Console.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
        }
    }
}
```



Accès aux champs simplifié

Héritage

Définir une hiérarchie de classes – Maximiser la réutilisation du code

new pour dire qu'on redéfinit la méthode

```
namespace Heritage
{
    /// <summary>
    /// Description of Voiture.
    /// </summary>
    public class Voiture
    {
        //marque
        private string marque;
        //modele
        private string modele;
        //puissance fiscale
        private int pfiscale;
        //constructeur
        public Voiture()
        {
        }
        //saisie des champs à la console
        public void saisie(){
            Console.Write("Marque : ");
            marque = Console.ReadLine();
            Console.Write("Modele : ");
            modele = Console.ReadLine();
            Console.Write("Puissance fiscale : ");
            pfiscale = int.Parse(Console.ReadLine());
        }

        public double taxe(){
            return 12.0 * (double)pfiscale;
        }
    }
}
```

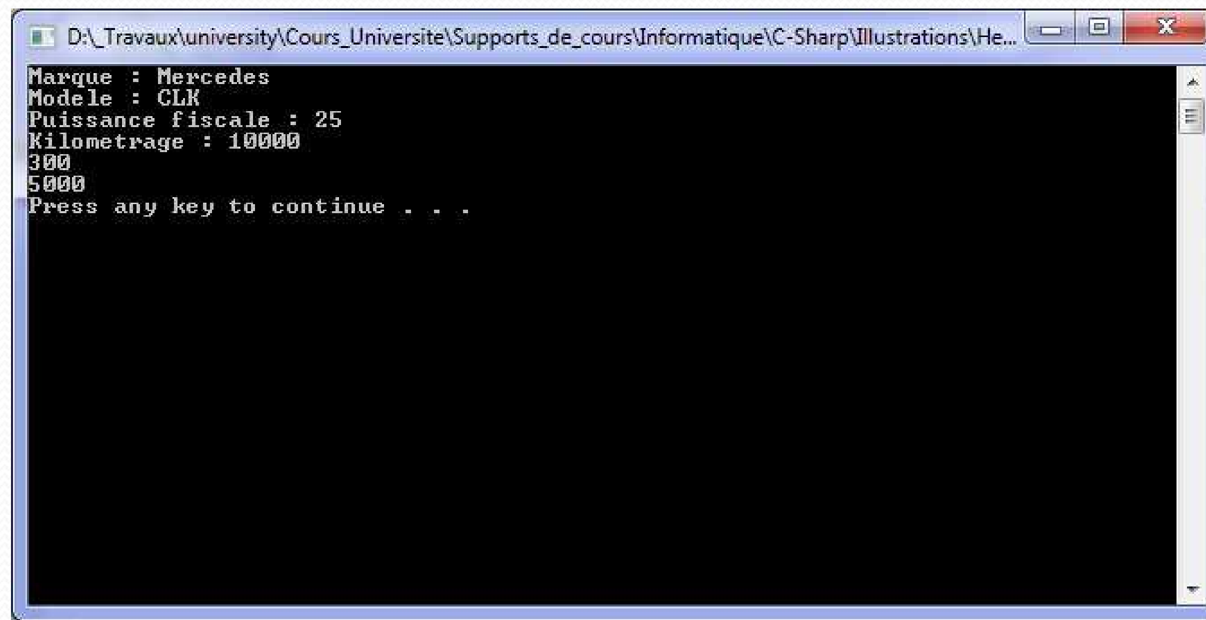
```
namespace Heritage
{
    /// <summary>
    /// Description of VoitureLocation.
    /// </summary>
    public class VoitureLocation : Voiture
    {

        //kilométrage
        private double km;

        public VoitureLocation()
        {
        }

        //surchage avec appel de la méthode de l'ancêtre
        public new void saisie(){
            base.saisie();
            Console.Write("Kilometrage : ");
            km = double.Parse(Console.ReadLine());
        }

        //nouvelle méthode
        public double prixLocation(){
            return 0.5 * km;
        }
    }
}
```

base pour faire appel à la méthode de l'ancêtre

Héritage

Instanciation

```
namespace Heritage
{
    class Program
    {
        public static void Main(string[] args)
        {

            VoitureLocation v = new VoitureLocation();

            v.saisie();

            Console.WriteLine(v.taxe());
            Console.WriteLine(v.prixLocation());

            Console.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
        }
    }
}
```



Polymorphisme

Le type déclaré (ancêtre) est différent du type réellement instancié (initialisé)

```
public class Voiture
{
    //marque
    private string marque;
    //modele
    private string modele;
    //puissance fiscale
    private int pfiscale;
    //constructeur
    public Voiture()
    {
    }
    //saisie des champs à la console
    public virtual void saisie()...

    //calcul de la taxe
    public double taxe()...
}
```

Méthode virtuelle, prend en charge le polymorphisme

```
public class VoitureLocation : Voiture
{

    //kilométrage
    private double km;

    public VoitureLocation()...

    //surcharge avec appel de la méthode de l'ancêtre
    public override void saisie()...

    //nouvelle méthode
    public double prixLocation()...
}
```

Surcharge tenant compte du polymorphisme

Type déclaré : Voiture

```
class Program
{
    public static void Main(string[] args)
    {

        Voiture v;

        v = new VoitureLocation();

        v.saisie();

        Console.WriteLine(v.taxe());
        Console.WriteLine((v as VoitureLocation).prixLocation());

        Console.Write("Press any key to continue . . . ");
        Console.ReadKey(true);
    }
}
```

Type instancié : VoitureLocation

Saisie de VoitureLocation car Saisie est virtuelle

Taxe de Voiture
Si Taxe est redéfinie chez VoitureLocation, elle est ignorée

PrixLocation de VoitureLocation
« Cast » explicite parce que la méthode n'existe pas chez Voiture

Classe abstraite

Classe dont certaines méthodes ne sont pas programmées

```
namespace Heritage
{
    /// <summary>
    /// Classe de base, gestion des employés d'une entreprise
    /// </summary>
    public abstract class Salarie
    {
        //nom de la personne
        private string nom;
        //aciennete
        protected int anciennete;
        //constructeur
        public Salarie()...

        public virtual void saisie(){
            Console.Write("Nom : ");
            nom = Console.ReadLine();
            Console.Write("Enciennete : ");
            anciennete = int.Parse(Console.ReadLine());
        }

        //son salaire n'est pas défini - abstract == virtuelle
        public abstract double salaire();

        //description du salarié
        public string description(){
            return this.GetType().ToString() + " : " + this.nom + " = " + salaire().ToString();
        }
    }

    //classe héritière
    public class Ouvrier:Salarie
    {
        public override double salaire(){
            return 1000.0 + 20 * anciennete;
        }
    }

    //classe heritiere
    public class Cadre:Salarie
    {
        private int grade;

        public override void saisie(){
            base.saisie();
            Console.Write("Grade : ");
            grade = int.Parse(Console.ReadLine());
        }

        public override double salaire(){
            return 1500.0 + 50 * grade + 10 * anciennete;
        }
    }
}
```

Une méthode est abstraite

→ Donc, la classe est abstraite

→ On ne peut pas créer une instance de cette classe !!!

Appel de la méthode abstraite dans une autre méthode de la classe abstraite

On peut quand même profiter des méthodes déjà programmées chez l'ancêtre

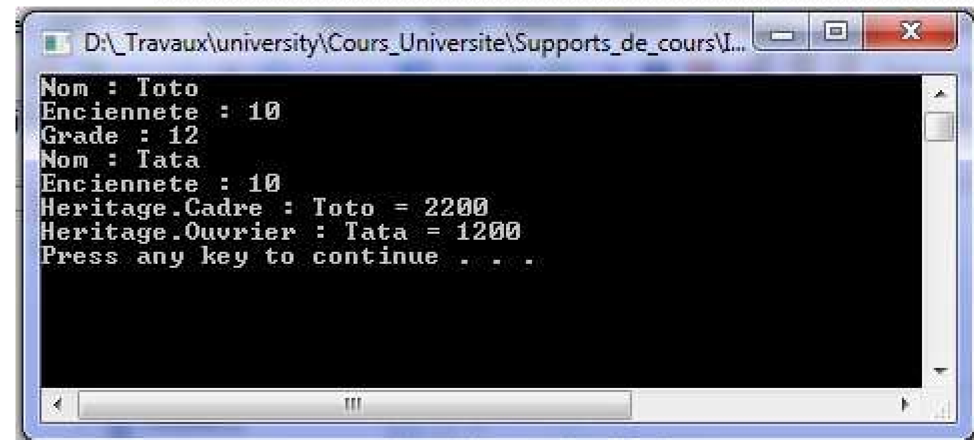
Liste polymorphe

Liste d'objets différents mais forcément héritiers du même ancêtre

```
using System;
using System.Collections.Generic;

namespace Heritage
{
    class Program
    {
        public static void Main(string[] args)
        {
            //liste de salariés
            List<Salarie> liste = new List<Salarie>();

            //variable tampon, du type de l'ancêtre
            Salarie s;

            //ajouter un cadre
            s = new Cadre();
            s.saisie();
            liste.Add(s);

            //ajouter un ouvrier
            s = new Ouvrier();
            s.saisie();
            liste.Add(s);

            //affichage
            foreach(Salarie p in liste)
                Console.WriteLine(p.description());


            Console.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
        }
    }
}
```

Pour accéder à la classe **List**

Notez au passage que List est une classe générique c.-à-d. paramétrée



FIN…

Les mêmes concepts sont – à peu de choses près – présents dans tous les langages de programmation objet (Java, Delphi, C++,…)