Les bases des classes en C# (suite)

Programmation C#

Les interfaces

Les interfaces – Définir une signature de classe

Interface = Classe abstraite pure
c.-à-d. on définit des signatures de méthodes, mais on ne les programme pas.
Ex. ISavable définit la signature « saveToFile »

```
/// <summary>
/// Description of ISavable.
/// </summary>
public interface ISavable
{
    //sauvegarde des informations dans un fichier
    void saveToFile(string nomFichier);
}
```

Quel intérêt ?

On peut écrire des méthodes de classes qui manipule des objets issues des interfaces -- *on dit qui « **implémentent** » les interfaces* – sans connaître précisément leur nature.

Ici on ne sait pas quel est le type réel de « objet », **mais on sait qu'il doit programmer les méthodes décrites dans ISavable**

```
/// <summary>
/// Description of ClassIOObject.
/// </summary>
public class ClassIOObject
{
    public ClassIOObject()
    {
    }

    public void sauvegarde(ISavable objet, string nomFichier){
        //vérififcation de l'existence et suppression éventuelle
        if (File.Exists(nomFichier) == true)
            File.Delete(nomFichier);
        //sauvegarde effective de l'objet ISavable
        //!\sans savoir ce que c'est vraiment
        //ni connaître le format de fichier
        objet.saveToFile(nomFichier);
    }
}
```

Les interfaces – Classes implémentant des interfaces

Voiture est une classe (héritière de Object par défaut).
Et **elle implémente les méthodes de ISavable**
Les informations sont sauvegardées dans un fichier texte.

```
// Description of Voiture
public class Voiture : ISavable
{
    public string modele;
    public string marque;
    public int pfiscale;

    public Voiture()
    {
    }

    //sauvegarde dans un fichier texte
    public void saveToFile(string nomFichier){
        StreamWriter sw = new StreamWriter(nomFichier);
        sw.WriteLine(modele);
        sw.WriteLine(marque);
        sw.WriteLine(pfiscale.ToString());
        sw.Close();
    }

}
```

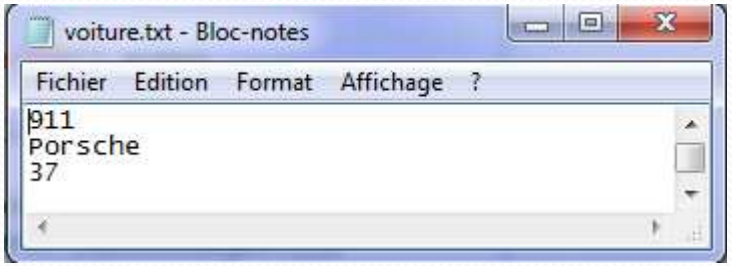
Personne est une classe (héritière de Object par défaut).
Et **elle implémente les méthodes de ISavable**
Les informations sont sauvegardées dans un fichier XML.

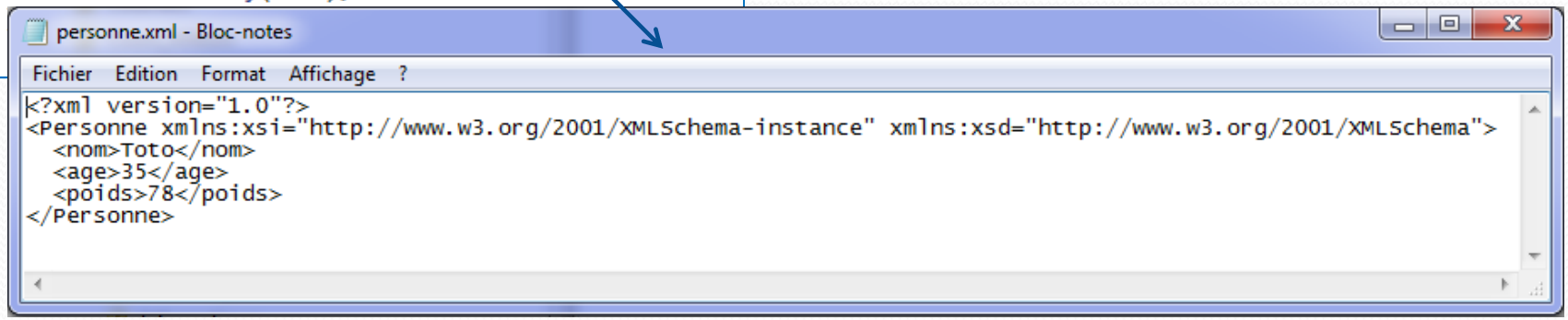
```
// Description of Personne.
[Serializable]
public class Personne : ISavable
{
    public string nom;
    public int age;
    public double poids;

    public Personne()
    {
    }

    //sauvegarde dans un flux XML
    public void saveToFile(string nomFichier){
        FileStream file = File.Open(nomFichier, FileMode.Create);
        XmlSerializer serializer = new XmlSerializer(typeof(Personne));
        serializer.Serialize(file, this);
        file.Close();
    }
}
```

Les interfaces – Instanciation et appel des méthodes

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

        //création de l'objet de sauvegarde
        ClassIOObject oSave = new ClassIOObject();

        //voiture
        Voiture v = new Voiture();
        v.marque = "Porsche";
        v.modele = "911";
        v.pfiscale = 37;
        //sauvegarde de voiture - fichier txt
        //appel standardisé
        oSave.sauvegarde(v,"voiture.txt");

        //personne
        Personne p = new Personne();
        p.nom = "Toto";
        p.poids = 78;
        p.age = 35;
        //sauvegarde de personne - fichier xml
        //appel standardisé
        oSave.sauvegarde(p,"personne.xml");

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

ClassIOObject peut sauver tout ce qu'on veut, pourvu que l'objet à sauvegarder ait implémenté les méthodes décrites dans l'interface ISavable [→ *saveToFile()*]





Les interfaces – Implémentation multiple

Une classe peut implémenter plusieurs interfaces pour être utilisé dans différents contextes.

```
/// <summary>
/// Description of ISavable.
/// </summary>
public interface ISavable
{
    //sauvegarde des informations dans un fichier
    void saveToFile(string nomFichier);
}
```

```
/// <summary>
/// Description of IChargeable.
/// </summary>
public interface IChargeable
{
    void loadFromFile(string nomFichier);
}
```

(1) Deux interfaces avec des signatures de méthodes pour les entrées-sorties.

(2) La classe implémente les deux interfaces.
Elle sait programmer les deux méthodes save et load.

```
// Description of Voiture.
public class Voiture : ISavable, IChargeable
{
    public string modele;
    public string marque;
    public int pfiscale;

    public Voiture()
    {
    }

    //sauvegarde dans un fichier texte
    public void saveToFile(string nomFichier){
        StreamWriter sw = new StreamWriter(nomFichier);
        sw.WriteLine(modele);
        sw.WriteLine(marque);
        sw.WriteLine(pfiscale.ToString());
        sw.Close();
    }

    //chargement à partir d'un fichier texte
    public void loadFromFile(string nomFichier){
        StreamReader sr = new StreamReader(nomFichier);
        modele = sr.ReadLine();
        marque = sr.ReadLine();
        pfiscale = int.Parse(sr.ReadLine());
        sr.Close();
    }

}
```

(3) La classe d'E/S se charge de sauver et charger sans savoir ce que c'est, pourvu que les signatures de méthodes soient respectées.

```
public class ClassIOObject
{
    public ClassIOObject()
    {
    }

    public void sauvegarde(ISavable objet, string nomFichier){
        //vérififcation de l'existence et suppression éventuelle
        if (File.Exists(nomFichier) == true)
            File.Delete(nomFichier);
        //sauvegarde effective de l'objet ISavable
        //!\sans savoir ce que c'est vraiment
        //ni connaître le format de fichier
        objet.saveToFile(nomFichier);
    }

    public bool chargement(IChargeable objet, string nomFichier){
        if (File.Exists(nomFichier) == false)
            return false;
        else
        {
            bool ok = true;
            //chargement sans connaître exactement la nature de l'objet
            try { objet.loadFromFile(nomFichier);}
            //renvoyer false si échec
            catch { ok = false; }
            return ok;
        }
    }
}
```

Les classes génériques

Classes génériques = classes paramétrées

La classe est paramétrée par **<T>**

(1) T est forcément une classe héritière de Object

(2) On peut (pas obligatoire mais souhaitable) la contraindre avec la clause « where » : dans notre exemple, on souhaite que seules les classes qui implémentent les méthodes de **IComparable** soient acceptées (*la méthode CompareTo(…) en l'occurrence*).

Remarques : *On peut contraindre avec plusieurs interfaces. On peut aussi contraindre en spécifiant la classe ancêtre à la racine de T (autre que Object, il ne faut pas non plus une classe « sealed » (sealed = spécifiquement sans héritier).*

```
// La classe est paramétrée par T
public class OpDifference<T> where T : IComparable
{
    private T val1;
    private T val2;

    //constructeur, envoi des valeurs
    public OpDifference(T o1, T o2)
    {
        val1 = o1;
        val2 = o2;
    }

    //dvision et renvoyer une valeur de type T
    public string comparaison(){
        //comparaison (renvoie un entier)
        int compare = val1.CompareTo(val2);
        //résultat
        string resultat = "egalite";
        if (compare < 0)
            resultat = val1.ToString() + " est inferieur a " + val2.ToString();
        else
            if (compare > 0)
                resultat = val1.ToString() + " est superieur a " + val2.ToString();
        return resultat;
    }
}
```

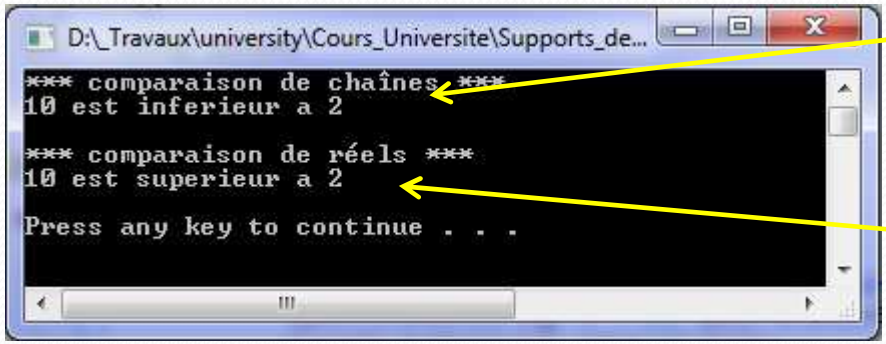
Sans la contrainte « where » pour T, cette instruction ne serait pas valide !

Classes génériques = classes paramétrées

Instanciation des objets

```
class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("*** comparaison de chaînes ***");
        OpDifference<String> d = new OpDifference<String>("10","2");
        Console.WriteLine(d.comparaison());


        Console.WriteLine("*** comparaison de réels ***");
        OpDifference<Double> e = new OpDifference<Double>(10,2);
        Console.WriteLine(e.comparaison());


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

On instancie deux fois la même classe, mais avec des paramètres différents.



Paramétrage avec String : « 10 » vs. « 2 »

Paramétrage avec Double : 10 vs. 2

Les méthodes statiques

Méthodes statiques associées aux classes

Ou comment ré-écrire de manière moderne le concept ancien des procédures et fonctions que l'on regroupe dans des modules (ex. les unités Pascal)

Dans une classe, certaines méthodes peuvent être statiques, d'autres non.
Les méthodes statiques peuvent être appelées sans avoir à créer une instance.
Elles ne peuvent manipuler que les champs statiques.

```
/// <summary>
/// Description of Calcul.
/// </summary>
public class Calcul
{

    private static double nombreE = 2.718281828;

    public Calcul()
    {
    }

    public static double exponentielle(double value){
        return Math.Pow(nombreE, value);
    }
}
```

this.NombreE n'a pas de sens parce que nombreE est un champ statique, associé à la classe et non pas à l'instance

Appel direct de la méthode sans passer par une instance. Comme les procédures et fonctions de l'ancien temps (C, Pascal, …).

```
class Program
{
    public static void Main(string[] args)
    {
        double v = Calcul.exponentielle(1.0);
        Console.WriteLine(v.ToString());

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

FIN…

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