

# RAPPELS

---

# Une classe

C'est un « type structuré » avec :

- champs privés
- propriétés
- constructeurs : but : initialiser les champs
- méthodes :
  - Héritées et à substituer : ToString, Equals, GetHashCode
  - spécifiques
- opérateurs surchargés

s'appuient

accèdent

# Une classe ou une structure ?

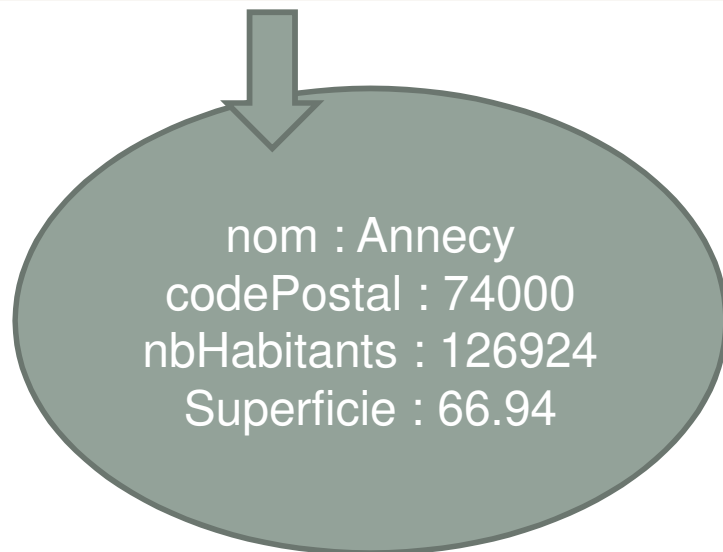
Structure et classe sont 2 notions très proches ... La différence :

- Structure : type valeur ( copie de valeurs lors de l'affectation )  
⇒ à préférer pour de petits éléments.
- Classe : type référence ( copie d'adresses lors de l'affectation )  
⇒ à préférer pour de grands éléments avec des données censées être modifiées après la création.

# Un objet

- C'est une instance de classe initialisée à l'aide :
  - Du mot clef **new**
  - D'un **constructeur qui initialise ses champs à l'aide des propriétés**

Ex : `Ville annecy = new Ville ("Annecy", "74000", 126924, 66.94)`



Une classe peut donner  
naissance à des milliers  
d'objets !

# Champs d'instances / Champs statiques

- d'instances par défaut
- statique si précisé : utile pour stocker une donnée commune à tous les objets de la classe ! Stockée une seule fois en mémoire, dissociée des objets !

```
class Commande
{
    private static double tarifLivraison = 3;

    public static double TarifLivraison
    {
        get { return Commande.tarifLivraison; }
        set { Commande.tarifLivraison= value; }
    }
}
```

```
Commande c1 = new Commande(1,"alimentation HP", 15.5);
Commande c2 = new Commande(1,"sacoche HP", 29.90);
Console.WriteLine(Commande.TarifLivraison);
Commande.TarifLivraison = 5;
```

On y accède depuis la classe

tarifLivraison :3

- quantite : 1  
- libelleArticle :  
sacocheHP  
- prixUnitaire :  
29,90

- quantite : 1  
- libelleArticle :  
alimentation HP  
- prixUnitaire :  
15,5

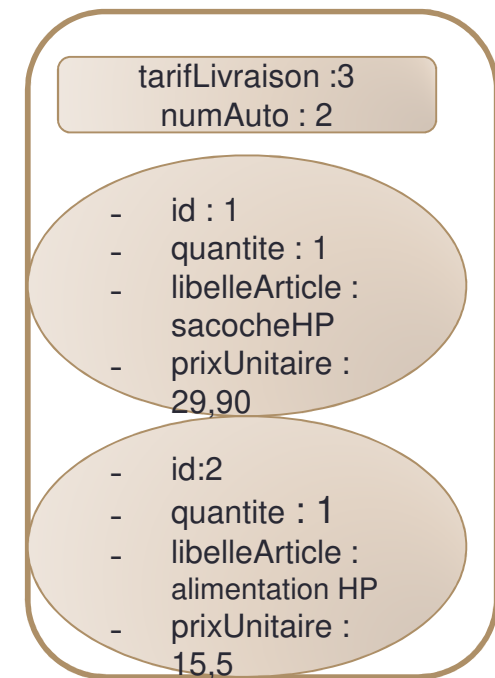
# Champs statiques

- Très utile aussi pour faire une donnée auto incrémentée : un identifiant.

```
class Commande
{
    private static int numAuto = 0;

    public static int NumAuto
    {
        get {
            Commande.numAuto++;
            return Commande.numAuto;
        }
        set { Commande.numAuto = value; }
    }

    public Commande(String libelle, double prixUnit, int quantite)
    {
        this.Id = Commande.NumAuto;
        this.LibelleArticle = libelle;
        this.PrixUnitaire = prixUnit;
        this.Quantite = quantite;
    }
}
```



# Constantes ( donc statiques )

constante = champ statique non modifiable => souvent publique.

- static read only (pour les objets )
- const ( pour les variables primitives )

```
class Commande  
{  
    public const double MONTANT_FRAIS_PORT_OFFERT = 50;  
}
```

```
Commande c1 = new Commande(1,"alimentation HP", 15.5);  
Commande c2 = new Commande(1,"sacoche HP", 29.90);  
Console.WriteLine(Commande. MONTANT_FRAIS_PORT_OFFERT);
```

On y accède depuis la classe

**MONTANT\_FRAIS\_PORT  
\_OFFERT : 50**

- quantite : 1  
- libelleArticle :  
sacocheHP  
- prixUnitaire :  
29,90

- quantite : 1  
- libelleArticle :  
alimentation HP  
- prixUnitaire :  
15,5

# Champs d'instance / Champs statiques

Champ d'instance	Champ statique
Donnée propre à un objet	Donnée commune à tous les objets
Appartient à l'objet	Appartient à la classe
Initialisée au sein du constructeur	Initialisée dès sa déclaration



## Accessibilité des champs

- Jusqu'à présent, on a fait des champs privés avec un accès en lecture/écriture via des propriétés publiques
- Mais on peut définir qu'ils soient d'instance ou statique:
  - Des champs privés avec un accès uniquement en lecture
  - Des champs totalement privés
  - Des champs publics
  - Des champs publics pour le projet, mais privé en dehors : Internal
  - Des champs protected : étudié plus tard

# Modificateur d'accès

- Méthodes, propriétés, champs, classes ont un modificateur d'accès :

Utilisable dans une	<code>public</code>	<u><code>internal</code></u>	<u><code>private</code></u>
Classe dans le projet	✓	✓	✗
Classe dans un autre projet	✓	✗	✗

- Remarque : Sans mot clef, par défaut, les champs, les propriétés, les méthodes sont privés et les classes sont internal

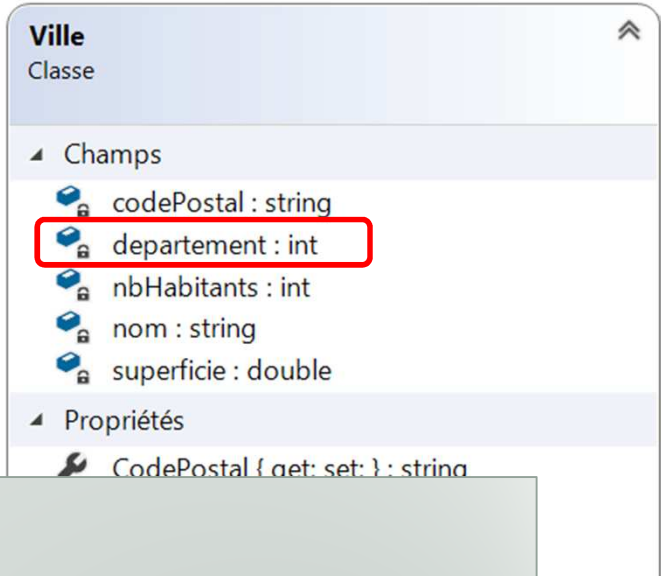
# Champs en lecture seule

- Très utiles pour stocker des données calculées. Remarque : on décide ici de stocker le département calculé à partir du codePostal.

```
public int Departement
{
    get
    {
        return this.departement;
    }

    private set
    {
        this.departement = value;
    }
}
```

```
public string CodePostal
{
    set
    {
        if (value==null)
            throw new ArgumentException("Le code postal est obligatoire ");
        //...
        this.Departement = int.Parse(value.Substring(0, 2));
    }
}
```



# Propriété calculée

Rem : il est possible aussi de faire une propriété basée uniquement sur un traitement. Mais dans ce cas, on ne stocke pas l'info du département

```
public int Departement
{
    get
    {
        return int.Parse(this.CodePostal.Substring(0, 2));
    }
}
```

# Propriété automatique

Lorsqu'aucune logique supplémentaire n'est ajoutée (aucun contrôle) dans les propriétés

=> pour une déclaration de propriété plus concise

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

Le champ nom privé sera  
généré automatiquement par  
le compilateur

# Constructeur

- C'est une méthode qui porte le même nom que la classe.
- Son but : initialiser les champs d'instance à l'aide des propriétés
- **Souvent surchargés : plusieurs signatures**

• Ex : `Ville annecy = new Ville ("Annecy", "74000", 126924, 66.94)`

# Surcharge du constructeur

- On peut surcharger pour rendre certains paramètres optionnels

Ex : on ajoute un champ privé (et une propriété) à Ville : **private bool estPrefecture;**  
**=> On peut alors définir 2 constructeurs**

// le constructeur « classique » : autant de paramètres que de champs

```
public Ville ( String nom, String codePostal, int nbHab, double surface, bool estPrefecture )  
{  
    this.Nom = nom ;  
    // .....  
    this.EstPrefecture = estPrefecture ;  
}
```

// un constructeur qui met false par défaut dans le champ estPrefecture

```
public Ville ( String nom, String codePostal, int nbHab, double surface )  
{  
    this.Nom = nom ;  
    // .....  
    this.EstPrefecture = false ;  
}
```

# Surcharge du constructeur

- Technique pour éviter le copier / coller
- **Favoriser la réutilisation**

// le constructeur « classique » : autant de paramètres que de champs

```
public Ville ( String nom, String codePostal, int nbHab, double surface, bool estPrefecture )  
{  
    this.Nom = nom ;  
    // .....  
    this.EstPrefecture = estPrefecture ;  
}
```

// un constructeur qui met false par défaut dans le champ estPrefecture

```
public Ville ( String nom, String codePostal, int nbHab, double surface) : this (nom, codePostal,  
nbHab, surface, false)  
{ } // pas de copier ici ! Pas de code !
```

Il est aussi possible de mettre une valeur par défaut et ainsi de ne pas surcharger !  
public Ville ( String nom, String codePostal, int nbHab, double surface, bool estPrefecture = false)



# Méthodes statiques ou d'instances

- Méthodes **d'instance** : s'applique sur un objet, toutes les infos contenues dans l'objet ne sont plus à passer en paramètre ( **A PREFERER !** )
- Méthodes de classe : static : s'applique sur une classe, il faut tout passer en paramètre

Ici : le même traitement défini de manière statique ou non

// méthode d'instance

```
public bool EstPlusDense(Ville v)
{
    return this.CalculDensite() > v.CalculDensite();
}
```

Ville annecy, chambéry ; ...  
If ( annecy.EstPlusDense ( chambéry))

// méthode de classe

```
public static bool EstPlusDense(Ville v1, Ville v2)
{
    return v1.CalculDensite() > v2.CalculDensite();
}
```

Ville annecy, chambéry ; ...  
If ( Ville.EstPlusDense (annecy, chambéry))

# De « fausses » classes

- On peut créer des classes uniquement pour ranger des méthodes statiques.

```
public class Read
{
    public static int ReadInt()
    {
        int res;
        while (Int32.TryParse(Console.ReadLine(), out res))
            Console.WriteLine("Erreur de saisie. Entier désiré.");

        return res;
    }

    public static double ReadDouble()
    {
        double res;
        while (Double.TryParse(Console.ReadLine(), out res))
            Console.WriteLine("Erreur de saisie. Réel désiré.");

        return res;
    }
}
```

# Enum

- Définir une énumération : nouveau type avec valeurs prédéfinies
  - Interne à une classe
  - Externe (dans un fichier à part)

```
public class Commande
{
    public enum Mode_Livraison { DOMICILE = 0, POINT_RELAIS = 1 };
    private Mode_Livraison livraison;
```

```
Commande c = new Commande( );
p.Livraison = Commande.Mode_Livraison.DOMICILE ;
```

- Et éviter cela :

```
public const int DOMICILE = 0 ;
public const int POINT_RELAIS = 1 ;
private int livraison;
public int Livraison
{
    get { return this.livraison; }
    set { if ( value != DOMICILE && value != POINT_RELAIS )
          throw new ArgumentException (« valeur inattendue »);
          this.livraison = value ;
    }
}
```

# COLLECTION DE DONNÉES

---

# Tableaux / Collections

- Pour gérer un ensemble d'informations de même type
- Choix entre tableaux et collections:
  - Tableau : taille fixe et multidimensionnel

=> utile pour des constantes ou données paramètres dont le nombre est fixé. Par exemple : jour de la semaine, tarifs

- Collection : taille variable

=> utile pour des listes d'informations variables : liste de produits, de clients ...

# Tableaux

- Taille fixe : possible de réallouer mais couteux !
- Initialisation lors de la déclaration : pas de taille

```
String [ ] jours = new String [ ] { "Lundi", "Mardi",  
"Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

- Déclaration puis initialisation

```
String [ ] jours = new String[7];  
jours[0] = "Lundi";  
jours[1] = "Mardi";  
jours[2] = "Mercredi";  
jours[3] = "Jeudi";  
jours[4] = "Vendredi";  
jours[5] = "Samedi";  
jours[6] = "Dimanche";
```

new = allocation  
mémoire contigue

# Tableaux

- Avec des objets

```
Ville [ ] lesVilles = new Ville[7];  
lesVilles[0] = new Ville ("Annecy", "74000", 126924, 66.94);  
lesVilles[1] = new Ville ("Chambéry", "73000", 58919, 20.99);  
...
```

new = allocation  
mémoire contigue

# Parcours de tableau

- Pour le lire :

```
foreach (String unJour in jours)
{
    Console.WriteLine(unJour);
}
```

```
for (int i=0 ; i< jours.Length ;i++)
{
    Console.WriteLine( jours[i] );
}
```

```
foreach (Ville uneVille in lesVilles)
{
    Console.WriteLine(uneVille);
}
```

unJour ou unVille : ce sont des copies des cases ( ici, des copies de références qui pointent vers les mêmes valeurs puisque ce sont des types références )



# Parcours de tableau

- Pour modifier des propriétés des objets contenus dans le tableau

```
foreach (Ville uneVille in lesVilles)
{
    uneVille.NbHabitants ++ ;
}
```

- Pour modifier l'objet dans sa globalité et donc potentiellement sa référence

```
for (int i=0 ; i < jours.Length ; i++)
{
    jours[i] = jours[i] + "s";
}
```

Interdit de modifier l'objet car cela modifierait la référence locale et non celle contenue dans le tableau

```
foreach (String unJour in jours)
{
    unJour = unJour + "s";
}
// (variable locale) string unJour
// CS1656: Impossible d'assigner à 'unJour', car il s'agit d'un 'variable d'itération foreach'
```

# Méthodes pour les tableaux

Méthodes statiques de classe Array rangées dans l'espace de nom System :

- int **IndexOf**( Array array, Object value ) : retourne la position de la 1ere occurrence de la valeur au sein du tableau
- T? **Find**<T> (T[] array, Predicate<T> match); retourne l'élément du tableau correspondant au prédicat
- void **Sort** (Array) : trie

```
String [ ] jours = new String [ ] { "Lundi", "Mardi", "Mercredi", ... "Dimanche" };  
int posMercredi = Array.IndexOf(jours, "Mercredi");  
Console.WriteLine("Mercredi c'est le jour " + (posMercredi+1));
```



```
Array.Sort(jours);
```

# Les collections

SortedList,  
SortedDictionary....

List<T>

Collection d'objets accessibles par index. Fournit des méthodes de recherche, de tri et de manipulation de listes.

Dictionary<TKey,TValue>

Collection de clés et de valeurs.

LinkedList<T>()

Collection d'objets doublement chaînés.

HashSet<T>

Ensemble d'objets sans double. Opérations ensemblistes mathématiques.

Stack<T>

Collection d'objets dernier entré, premier sorti => LIFO(Last in, first out)

Queue<T>

Collection d'objets premier entré, premier sorti. file d'attente sous la forme d'un tableau circulaire. Les objets stockés dans un Queue sont insérés à une extrémité et supprimés de l'autre.  
=> FIFO(First in first out)


# Listes : List <T>

<T> signifie qu'on met  
tout type

- Initialisation lors de la déclaration : taille non obligatoire

```
List<String> prenom = new List<String> ();  
ou  
List<String> prenom = new List<String> (10);  
Ou  
List<String> prenom = new List<String>{ "Franck", "Aimerick", ... };
```

- Déclaration puis initialisation à l'aide de la méthode Add:

```
List<String> prenom = new List<String>(10); // création de la liste  
  
prenom.Add("Franck");  
prenom.Add("Aimerick");  
prenom.Add("Elodie");
```

# Listes : list <T>

- Avec des objets

```
List<Ville> lesVilles = new List<Ville>(); // création de la liste  
  
lesVilles.add( new Ville ("Annecy", "74000", 126924, 66.94) );  
lesVilles.add ( new Ville ("Chambéry", "73000", 58919, 20.99) );
```

# Parcours de liste

- Pour le lire :

```
List<String> prenom; //...  
foreach (String prenom in prenom)  
    Console.WriteLine(prenom);
```

```
for (int i = 0; i < prenom.Count; i++)  
    Console.WriteLine( prenom[i] );
```

```
List<Ville> lesVilles ; //...  
foreach (Ville v in lesVilles)  
    Console.WriteLine(v);
```

Attention : impossible de  
supprimer ou d'ajouter des  
éléments pendant un  
parcours de liste

# Parcours de listes

- Pour modifier des propriétés des objets contenus dans le tableau

```
foreach (Ville uneVille in lesVilles)
{
    uneVille.NbHabitants ++ ;
}
```

- Pour modifier l'objet dans sa globalité et donc potentiellement sa référence

```
for (int i=0 ; i < lesVilles.Count ; i++)
{
    lesVilles[i] = new Ville ( lesVilles [i].Nom,...);
}
```

Ici, en recréant une ville on donne une nouvelle référence à la case qui ne pointera plus sur la zone mémoire initiale

# Méthodes pour les listes : list <T>

Méthodes pour les objets de classe List :

- void Add( T item )
- void Insert( int index, T item )
- void Sort () : trie

...

```
List<String> prenom = new List<String>(); // création de la liste  
prenom.Add("Franck");  
prenom.Add("Aimerick");  
prenom.Add("Elodie");
```



```
prenom.Sort();
```

Attention, Sort doit s'appuyer  
sur une méthode de  
comparaison



# Méthodes pour les listes : list <T>

Méthodes pour les objets de classe List :

- int IndexOf( T value ) : retourne la position de la 1ere occurrence de la valeur au sein du tableau
- int LastIndexOf( T value ) : retourne l'index de la dernière occurrence de la valeur au sein du tableau
- T? **Find**<T> ( Predicate<T> match); retourne l'élément de la liste correspondant au prédicat
- List<T> **FindAll** (Predicate<T> match); retourne une liste constituée des éléments de la liste correspondants au prédicat

```
List<Ville> lesVilles = new List<Ville>(); // création de la liste
Ville villeAnnecy = lesVilles.Find ( v => v.Nom == "Annecy");
List<Ville> lesGrandesVilles = lesVilles.FindAll ( v => v.Superficie > 100000);
```

Attention, IndexOf, Find et FindAll s'appuient sur la méthode Equals du type de la liste

# Trier une liste

- Définir la méthode de comparaison dans la classe
- Trier

```
public class Employee: IComparable<Employee>
{
    public string name;
    public int age;
```

```
List<Employee> employees ;
....
employees.Sort();
```

```
    public int CompareTo(Object o)
    {
        if ( obj ==null || !(obj is Employee))
            throw new ArgumentException("Le paramètre doit
            Employee other = (Employee)obj;
            return this.Age.CompareTo( other.Age);
    }
```

Un seul tri possible

Value	Condition
Inférieure à zéro	Cette instance précède <code>value</code> .
Zéro	Cette instance a la même position dans l'ordre de tri que <code>value</code> .
Supérieure à zéro	Cette instance suit <code>value</code> .

# Trier une liste

- Définir des méthodes de comparaison dans la classe
- Trier avec la méthode de son choix

```
public class Employee  
{  
    public string name;  
    public int age;
```

```
List<Employee> employees ;  
....  
employees.Sort(Employee.CompareParAge);
```

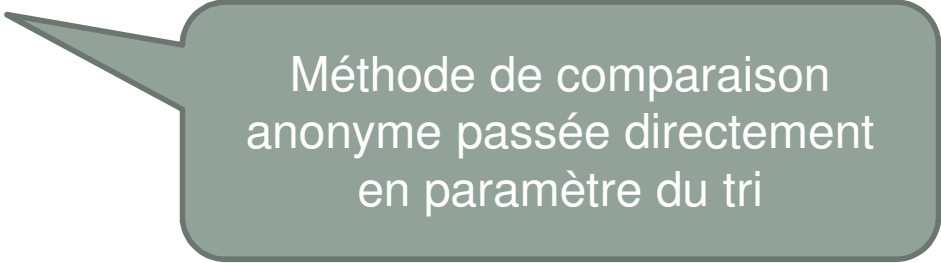
```
public static int CompareParAge(Employee e1, Employee e2)  
{  
    return e1.Age.CompareTo( e2.Age);  
}
```

Value	Condition
Inférieure à zéro	Cette instance précède <code>value</code> .
Zéro	Cette instance a la même position dans l'ordre de tri que <code>value</code> .
Supérieure à zéro	Cette instance suit <code>value</code> .

# Trier une liste

- Passer en paramètre la méthode de comparaison

```
List<Employee> employees ;  
....  
employees.Sort(delegate(Employee x, Employee y)  
                { return x.age.CompareTo(y.age); });
```



Méthode de comparaison  
anonyme passée directement  
en paramètre du tri

- Encore plus concis avec des expressions lambda

```
employees.Sort((x, y) => x.age.CompareTo(y.age));
```

# Dictionnaires : Dictionary <Tkey, TValue>

- Initialisation lors de la déclaration : pensez à estimer une taille

```
Dictionary<String, Ville> lesVilles = new Dictionary<String, Ville>(100);
```

- Déclaration puis initialisation à l'aide de la méthode Add:

```
Dictionary< String, Ville> lesVilles = new Dictionary< String, Ville>(100);  
lesVilles.Add("74000", new Ville ("Annecy", "74000",126924, 66.94) );  
lesVilles.Add("73000", new Ville ("Chambéry", "73000",58919, 20.99) );
```

74000	73000
74000	73000
Annecy	Chambéry
126924	58919
66.94	20.99

# Parcours de dictionnaire

- Pour le lire :

## // Pour lire les valeurs

```
foreach ( Ville uneVille in lesVilles.Values)
    { Console.WriteLine(uneVille); }
```

## // Pour lire les clefs

```
foreach ( String unCodePostal in lesVilles.Keys)
    { Console.WriteLine(unCodePostal);}
```

## // Pour lire le couple clef, valeur

```
foreach ( KeyValuePair< String, Ville> uneVille in lesVilles)
    { Console.WriteLine(uneVille.Key + "=>" + uneVille.Value);
  }
```

# Méthodes pour les dictionnaires

Méthodes pour les objets de classe Dictionary :

- **bool ContainsKey (TKey key)**
- **bool Remove(Tkey key)**
- **bool ContainsValue (TValue value);**
- ...

Accès à un élément grâce à la clef :

Ex: `Console.WriteLine(lesVilles["74000"]);`

# PERSISTANCE DES DONNÉES

---

Ou comment sérialiser les objets (les données) ...



# Travailler avec des fichiers

- On peut écrire et lire des données dans des fichiers

- .csv
- .txt

```
300;Marc  
280;Lucie  
245;Rafaella
```

Format plus ancien, facile  
à lire avec un tableur

- On peut sérialiser et désérialiser des objets dans des fichiers :

- .json
- .xml

```
[{"NbPoints":300,"Prenom":"Marc"}, {"NbPoints":280,"Prenom":"Lucie"}]  
  
<?xml version="1.0" encoding="utf-8"?>  
<ArrayOfScore xmlns:xsi="http://www.w3  
  <Score>  
    <NbPoints>500</NbPoints>  
    <Prenom>Léa</Prenom>  
  </Score>  
  <Score>  
    <NbPoints>300</NbPoints>  
    <Prenom>Marc</Prenom>  
  </Score>
```

- Avec des package additionnels, on peut travailler avec des fichiers au format .xls ....

=> Travailler avec un fichier : ce n'est pas travailler avec une base de données. On charge tout pour tout réécrire.

# Exceptions possibles

- Travailler avec des fichiers peut générer plein d'exceptions !

## Exceptions

### `UnauthorizedAccessException`

L'accès est refusé.

### `ArgumentException`

Le `path` est une chaîne vide ("").

- ou - `path` contient le nom d'un périphérique système (com1, com2, etc.).

### `ArgumentNullException`

`path` a la valeur `null`.

### `DirectoryNotFoundException`

Le chemin spécifié n'est pas valide (par exemple, il est sur un lecteur non mappé).

### `PathTooLongException`

Le chemin et/ou le nom de fichier spécifiés dépassent la longueur maximale définie par le système.

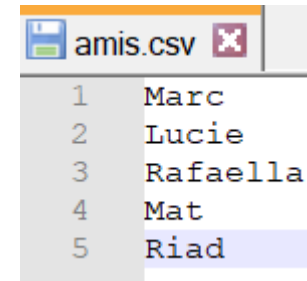
On utilisera alors des blocs  
`try { } catch { }`

# Quelques unes des classes utiles

- **FileStream** : utile pour fichier binaire: pour lire et écrire des séquences d'octets.
- **File** : utile pour copie, déplacement, changement de nom, création, ouverture, suppression et ajout de fichier, lecture globale.
- **StreamReader/StreamWriter** : ( s'appuie sur FileStream ) : pour lire et écrire dans des fichiers textes, gère l'encodage..

# Lire un fichier : chargement

- J'instancie un **objet** pour avoir un accès en lecture vers le fichier. Cela positionne un curseur en début de fichier.
- Tant que le curseur n'est pas **en fin de fichier**
  - Je **lis la ligne** en cours (et la stocke en mémoire)
- Je **ferme** l'accès à ce fichier



```
List<String> lesAmis = new List<string>();
try
{
    StreamReader reader = new StreamReader("amis.csv");
    while (!reader.EndOfStream)
    {
        String ami = reader.ReadLine();
        lesAmis.Add(ami);
    }
    reader.Close();
} catch (Exception e) { Console.WriteLine(e); }
```

Attention : emplacement par défaut bin où se trouve l'exe.

Demo1 > bin > Debug

Nom

amis.csv

Demo1.dll

Demo1.exe

# Lire un fichier : chargement

- Les lignes du fichier peuvent être plus structurées.
- Il faut alors extraire les valeurs pour les stocker dans des objets.

```
List<Personne> lesAmis = new List<Personne>();  
try  
{  
    StreamReader reader = new StreamReader("amis.csv");  
    while (!reader.EndOfStream)  
    {  
        String ligne = reader.ReadLine();  
        String[] values = ligne.Split(";");  
        lesAmis.Add(new Personne(values[0], values[1], values[2]));  
    }  
    reader.Close();  
} catch (Exception e) { Console.WriteLine(e); }
```

```
1 | Marc;Rabip;0607080921  
2 | Lucie;Malro;0708023445  
3 | Rafaela;Paulu;0689234500
```

0	1	2
Marc	Rabip	0607080921

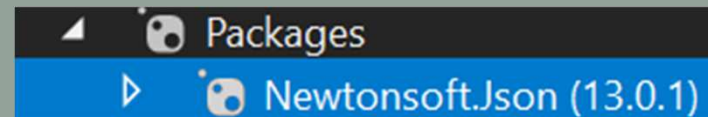
# Ecrire dans un fichier : Sauvegarde

- J'instancie un **objet** pour avoir un accès en écriture dans un fichier.
- Tant que j'ai des infos à écrire
  - J'écris une nouvelle ligne
- Je **ferme** l'accès à ce fichier

```
List<String> lesAmis = new List<string>();  
// ajout d'éléments dans la liste...  
try  
{  
    StreamWriter writer = new StreamWriter("amis.csv");  
    foreach (String ami in lesAmis)  
    {  
        writer.WriteLine(ami);  
    }  
    writer.Close();  
} catch (Exception e) { Console.WriteLine(e); }
```

# Désérialiser un format json

- Je récupère tout le contenu du fichier dans une chaîne de caractères
- Je **désérialise** la chaîne formatée en json = j'extrais les données d'un flux et les convertis dans un format utile à l'application.



```
List<Personne> lesAmis ;  
try  
{  
    String contenuFichier = File.ReadAllText("amis.json");  
    lesAmis = JsonConvert.DeserializeObject<List<Personne>>(contenuFichier);  
}  
catch (Exception e) { Console.WriteLine(e); }
```

## Attention :

- Les propriétés doivent porter le même nom que dans le fichier json, et doivent être publiques
- En cas de surcharge de constructeur, la classe doit avoir un constructeur par défaut !

# Sérialiser des objets en json

- Je **sérialise** les objets en une chaîne json.
- J'écris tous dans le fichier

```
List<Personne> lesAmis ;  
// ...  
try  
{  
    string result = JsonConvert.SerializeObject(lesAmis ,Formatting.Indented) ;  
    File.WriteAllText("amis.json", result);  
}  
catch (Exception e) { Console.WriteLine(e); }
```



# Désérialiser un format xml

- J'instancie un **objet** capable de désérialiser du xml
- J'instancie un **objet** pour avoir un accès en lecture du fichier.
- Je **désérialise** le document xml
- Je **ferme** l'accès à ce fichier

La classe à sérialiser doit être public et doit avoir un constructeur par défaut

```
List<Personne> lesAmis ;
XmlSerializer xs = new XmlSerializer(typeof(List< Personne >));
try
{
    StreamReader reader = new StreamReader("amis.xml");
    lesAmis = xs.Deserialize(reader) as List<Personne>;
    reader.Close();
}
catch (Exception e) { Console.WriteLine(e); }
```

# Sérialiser des objets en xml

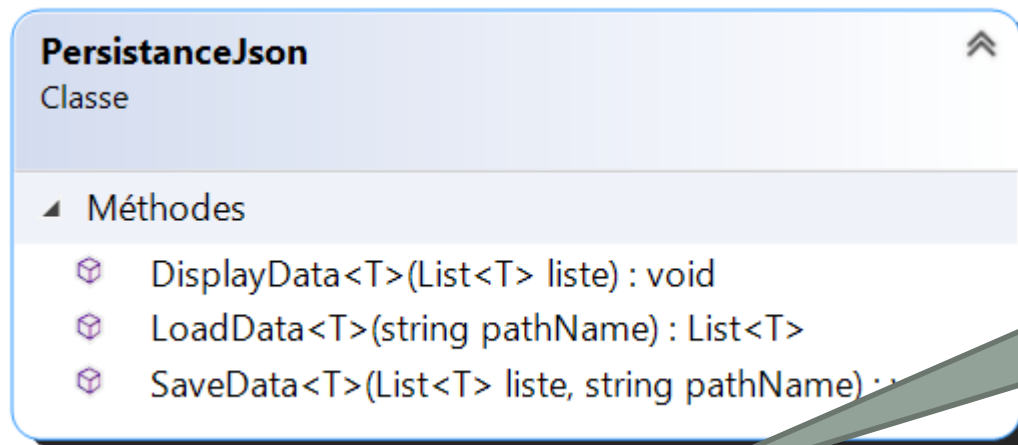
- J'instancie un objet capable de sérialiser en xml
- J'instancie un objet pour avoir un accès en écriture au fichier.
- Je sérialise les objets dans le fichier xml
- Je ferme l'accès à ce fichier

La classe à sérialiser doit être public et doit avoir un constructeur par défaut

```
List<Personne> lesAmis ;  
// ...  
XmlSerializer xs = new XmlSerializer(typeof(List< Personne >));  
try  
{  
    StreamWriter writer = new StreamWriter("amis.xml");  
    xs.Serialize(writer, lesAmis);  
    writer.Close();  
}  
catch (Exception e) { Console.WriteLine(e); }
```

# Faire un code générique

- Il est possible de faire du code générique pour un format:



Il faut annoncer lors de la déclaration de la méthode qu'elle s'appuie sur <T>

```
public static void SaveData<T>(List<T> liste, String pathName)
{
    try
    {
        string result = JsonConvert.SerializeObject(liste, Formatting.Indented);
        File.WriteAllText(pathName, result);
    }
    catch (Exception e) { throw ; }
}
```

Il faut renvoyer l'éventuelle exception vers l'appelant

# A suivre

- La persistance avec les BD....