E] Les tableaux et énumérations

1) Enumérations

Une énumération est un type de données dont le domaine de valeurs est un ensemble de constantes entières

Ex: liste de genre musical Rock, Classique, Soul, Jazz

```
enum Genres { Rock, Classique, Soul, Jazz };
```

Codées par des entiers consécutifs 0 1 2 3

Une variable peut être déclarée comme prenant valeur dans l'énumération

```
Genres unGenre = Genres.Rock;
```

On peut la comparer

```
if ( unGenre != Genres.Soul)
    Console.WriteLine (" c'est nul ");
```

On peut boucler sur les valeurs de l'énumération

```
Foreach ( Genres g in Enum.GetValues (unGenre.GetType()))
{
    Console.WriteLine (g);
}
```

Enum ⇒ System.Enum méthode GetValues qui permet d'obtenir les valeurs d'un type enuméré que l'on passe en paramètre -

unGenre.GetType() donne l'objet Type de l'énumération Genres

```
Foreach (int g in Enum.GetValues (typeof(Genres)))
```

La boucle se fera sur les entiers.

2) Les tableaux

Syntaxe

unidimensionnel : `<type> [] <identificateur> = new <type>[<taille>];`

Ex :
```
int [] tabEntiers = new int [4];
char[] tabChars = new char [10];
string[] tabChaines = new string[7];
int tailleTableau = 12;
char[] tabChar = new char [tailleTableau];
```

multi : `<type> [,] <identificateur> = new <type> [<taille 1>, <taille 2>];`

(autant de virgules que de dimensions)

Ex :
```
int[,] tabEntiers2d = new int [4, 4];
double[,,] tabDoubles3d = new double [6,6,6];
```

Méthodes et prop

- `monTab.Length` renvoie un entier indiquant la taille
- `Array.Sort(monTab)` trie
- `Array.Reverse(...)` inverse l'ordre

Foreach

```
string[] noms = new string[] { "Alex", "Toto", "Jean" }
Foreach (string n in noms)
{
    ...
}
```

III Les classes

But définir de nouveaux types spécifiques aux besoins de l'application

⚠ objet ≠ classe la classe est le moule, l'objet un exemplaire – (on dit une instance)

1) Définition

La syntaxe est <public, (private, protected)> class <identificateur> {
<public, private, protected> donnée ou méthode ou propriété ;
}

ex:
```
public class Personne
{
    // attributs ou données
    private string nom;
    private string prenom;
    private int age;
    ...
}
```

Les membres d'une classe sont
- les données ou attributs
- les méthodes (fonctions qui manipulent les données)
- les propriétés méthodes servant à lire ou connaître la valeur d'un attribut

Ils sont de type : private : accessible par les méthodes internes de la classe
protected : ——— ou d'un dérivé
public : accessible par toutes les méthodes –

Les données sont privées pour en restreindre les possibilités de manipulation.

2) Construction : méthode Initialise() et mot-clé new

```
class Personne {
    ...
    public void Initialise (string N, string P, int A)
    {
        this.prenom = P;
        this.nom = N;
        this.age = A;
    }
```

```
    public void Identifie () {
        Console.WriteLine ("[{0}, {1}, {2}]", prenom, nom, age);
    }
}
```

- Comme les données sont privées on ne peut pas faire

```
Personne p1;
p1.nom = "Dupond";
```

- Si on saisit le code :
```
Personne p1;
p1.Initialise ("Jean", "Dupond", 24);
```

ces instructions sont illégales car Initialise() est public.
→ incorrecte car p1 définie une référence à un objet qui n'existe pas encore.

- le code correct est
```
Personne p1 = new Personne();
p1.Initialise ("Jean", "Dupond", 24);
```

après la première ligne les attributs nom et prenom sont de valeurs "null" et age vaut 0. La deuxième ligne initialise ces paramètres.

- Mot-clé this

this.prenom = p signifie que l'attribut prenom de l'objet courant reçoit la valeur p. L'objet courant est visible dans la ligne d'appel à la méthode : p1.Initialise ...
p1 = objet courant

- Une autre méthode Initialise

```
public void Initialise (Personne p)
{
    prenom = p.prenom;
    nom = p.nom;
    age = p.age;
}
```

→ deux méthodes Initialise car elles prennent deux jeux de paramètres différents
→ Initialise a accès aux données de p car un objet o1 d'une classe c a toujours accès aux attributs des objets de la même classe c.

```
Class Program {
    static void Main () {
        Personne p1 = new Personne();
        p1.Initialise ("Jean", "Dupond", 24);
        p1.Identifie();
        Personne p2 = new Personne();
        p2.Initialise (p1);
        p2.Identifie();
    }
}
```

[Jean, Dupond, 24]
[Jean, Dupond, 24]

Constructeurs

Un constructeur est une méthode qui porte le nom de la classe et qui est appelé lors de la création de l'objet. On s'en sert pour l'initialisation. Cette méthode ne renvoie aucun résultat (void)

Si une classe C a un constructeur qui prend n arguments on pourra utiliser

```
C objet = new C(arg1, ..., argn);
```

ou

```
C objet;
objet = new C(arg1, ..., argn);
```

A partir des deux méthodes Initialise on peut créer deux constructeurs

```
public Personne(string P, string N, int A)
{ Initialise(P, N, A); }
public Personne(Personne p)
{ Initialise(p); }
```

```
Personne p1 = new Personne("Jean", "Dupond", 20);
Personne p2 = new Personne(p1);
```

← constructeur d'instance

3) Les références d'objet

```
Personne p1 = new Personne("Jean", "Dupond", 30);
——      p2 = p1;
p1.Identifie();   [J, D, 24]
p2.——;            "
Personne p3 = new Personne(p1);
p1.Initialise("Marcel", "Bertrand", 45);
p1, p2, p3 .Identifie   [J, D, 24]
                        [J, D, 24]
                        [M, B, 45]
```

4) Lecture et écriture des attributs

```
// accesseurs
public string GetPrenom()
{
    return prenom;
}
```

```
// modificateurs
public void SetPrenom(string P)
{
    Rg.prenom = P;
}
```

Les propriétés

Elles permettent de manipuler des attributs comme s'ils étaient privés

```
public <Type> <Propriété>
{
    get { ... }
    set { ... }
}
```

- Type est le type de l'attribut
- set reçoit un paramètre appelé value qu'elle affecte à l'attribut qu'elle gère (on en profite pour faire des vérifications)

Ex :
```
Personne p = new Personne("Jean", "Dupond", 26);
Console.WriteLine("p = [" + p.Prenom + ", " + p.Nom + ", " + p.Age + "]");
p.Age = 56;
```

avec dans le fichier classe la propriété suivante

```
public string Nom {
    get { return nom; }
    set {
        if (value == null || value.Trim().Length == 0) {
            throw new Exception("nom (" + value + ") invalide");
        } else {
            nom = value;
        }
    }
}
```

Pour age
```
Public string Age {
    get { return age; }
    set {
        if (value >= 0) {
            age = value;
        } else {
            throw new Exception("age (" + value + ") invalide");
        }
    }
}
```

- On peut les utiliser avec un constructeur

  `Classe objet = new Classe(...) { Prop1 = val1, ... }`

- (Si pas de traitement on peut utiliser la prop automatique)

5) Les méthodes et attributs de classe

Les méthodes et attributs seront liés au moule et non aux instances

Ex: connaître le nbre d'objet d'une ~~un~~ classe

```
private static long nbPersonne;          ← declaration

public static long NbPersonne {          ← Propriété
    get { return nbPersonne; }
}

nbPersonne ++;                           ← à rajouter dans tous les constructeurs
```

IV L'Héritage

→ Personne abstraction d'une classe ancêtre.

→ Ex: Enseignant est une personne : . il reprend les attributs de la classe personne
. on lui ajoute d'autres attributs.

```
class Enseignant : Personne {
   classe dérivée / fille        ↳ classe parent / mère

    private int section;  ( section n° de la section, discipline)

    public Enseignant ( string prenom, string nom, int age, int section)
                : base ( prenom, nom, age) {

        Section = section; // via la propriété
        Console.WriteLine (" Constructeur Enseignant (string, string, int, int) ");
    }

    public int Section {
        get { return section; }
        set { section = value; }
    }

}
```

constructeur

propriété

Constructeurs
- Une classe fille n'hérite pas des constructeurs.
- via l'instruction : base ( ) on peut récupérer le constructeur mère.
- On pourrait réécrire un constructeur complet :

```
public Enseignant (string prenom, string nom, int age, int section)
{
    this.nom = nom;
    ...
}
```

illégal car le champs nom est privé. Il faudrait penser à mettre protected

Méthodes ou propriétés On peut écrire

```
Enseignant e1 = new Enseignant ("Jean", "Dupont", 24, 26)
Enseignant.Identite
```

(ou dans prénom on a défini proj

```
public Identite {
    get { return String.Format("[{0},{1},{2}]",
                nom, prenom, age); }
}
```

)

Enseignant n'a pas de proj Identite mais elle hérite celle de sa classe mère
Pb: il manque le champs section : on redéfinit la proj.

```
public new string Identite {
    redefinition
    get { return String.Format("Enseignant [{0},{1}]", base.Identite, section); }
}
```

Polymorphisme

Soit une lignée de classes : $C_0 \leftarrow C_1 \leftarrow C_2 \dots \leftarrow C_n$

O_i classe C_i, O_j classe C_j $\quad O_i = O_j \quad j > i$

fait que O_i est une référence à l'objet de type C_i inclus dans O_j

```
public static void Affiche (Personne p)
{
    ...
}
```

On peut utiliser Affiche (e) ou Enseignant e; -

Redéfinition et polymorphisme

```
public static void Affiche (Personne p)
{
    Console.WriteLine (p.Identite);
}
```

Si on fait `Enseignant e = new Enseignant ( "Jean", "Dupond", 24, 26);`
`Affiche (e);`

va donner [Jean, Dupond, 24]; c'est la propriété Identité de Personne qui a été utilisée.

Pour avoir identité de Enseignant il faut déclarer la propriété comme virtuelle dans la classe Personne.

```
public virtual string Identite {
    get { return String.Format ("[{0}, {1}, {2}]", prenom, nom, age); }
}
```

Ce mot-clé peut s'appliquer aux méthodes. Les fils classes qui redéfinissent cet élément doivent utiliser le mot-clé override et non new

```
public override string Identite {
    get { return String.Format ("Enseignant[{0}, {1}]", base.Identite, section); }
}
```

La fonction affiche utilisera alors la bonne propriété en fonction de l'argument même si la déclaration reste la même.

V Surcharge d'opérateur

1) Introduction

Considérons l'instruction `op1 + op2`. On peut redéfinir l'opération + pour une classe. Si l'opérande op1 est de classe C1, il faut définir une méthode statique dans la classe C1:

```
public static [type] operator + (c1 op1, c2 op2);
```

le type est important car si on a `op1 + op2 + op3` → `(op1 + op2) + op3` et si C1.operator+ est de type C1 on pourra utiliser l'associativité.

On peut redéfinir les opérateurs unitaires ++ et --

△ || == et != doivent être redéfinis en même temps
&&, ||, [], (), +=, -= ne peuvent être redéfinis.

25 Ex | Distribuer le code de ~~ArrayList~~ ListeDePersonnes
| Projeter

16 : ListeDePersonnes dérive d'ArrayList
18-13 : Redéfinition de +
112 : operator+ renvoie un ListeDePersonnes afin de pouvoir utiliser $l1 + p2 + p3$.

du coup dans le main

```
ListeDePersonnes ... l = new
Personne p1, p2, p3
l = l + p1 + p2;
l = l + p3
```

VI Definir un indexeur

Si l est un objet ListeDePersonnes on veut pouvoir utiliser l[i] en lecture (Personne p = l[i]) ou en écriture (l[i] = new Personne(...)).
Or comme ListeDePersonne dérive d'ArrayList cela ne fonctionne pas.
On rajoute une propriété

```
public Personne this[int i] {
    get {...}
    set {...}
}
```

Comme la classe ArrayList a un indexeur de type

```
public Object this[int i] {
    ...
}
```

il y a un conflit. donc

```
public new Personne this[int i] {
{
    get { return (Personne) base[i]; }
    set { base[i] = value; }
}
```

Pour indexer via un nom :

```
public int this [string nom] {
    get {
        for (int = 0 ; i < Count ; i++)
        {
            if(((Personne) base[i]).Nom == nom)
                return i;
        }
        return -1;
    }
}
```

VII Les classes abstraites

Une classe abstraite est une classe qui ne pourra pas être instanciée
Il faudra créer des classes dérivées - On l'utilise pour factoriser
le code d'une lignée de classe -

```
abstract class Utilisateur {
    private string login;
    private string mdp;
    private string role;
    public Utilisateur (string L, string MdP)
    {
        this.login = L;
        this.mdp = MdP;
        role = Identifie();
        if (role == null)
            throw new Exception(UtilisateurInconnu) (String.Format("{0}, {1}"
                                                    , L, MdP);
    }
    public override string ToString(){
        return String.Format("{0}, {1}, {2}", login, mdp, role);
    }
    abstract public string Identifie();
}
```

→ à remonter de la string

Les classes filles devront fournir la méthode Identifie()

```
class Admin : Utilisateur{
    public Admin (string L, string MdP) : base (L, MdP) {}
    public override string Identifie() {
        // identification externe
        ...
        return "admin";
    }
}
class Observateur
    ...
    // identification SGBD
    ...

class Inconnu : ...
    ...
    return null;
```

Au final observateur et Administrateur sont construits par le même constructeur -

VIII Les structures

La structure en C# est proche du C et est très semblable à la classe.

```
struct  Nom Structure
{
    // attributs
    ...
    // prop
    ...
    // methode
    ...
    // constructeurs
}
```

Il n'y a pas d'héritage avec les structures.

```
class  c Personne
struct  SPersonne
{
    public string Nom;
    public int Age;
}
```

```
Sp1.Nom = "Paul"
Sp1.Age = 24
Sp2 = Sp1
Sp2.Nom = "Henri"
```

1 : Paul, 24
Henri, 24

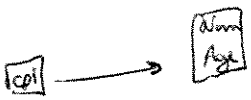
```
Cp1...
Cp2
Cp2 = Cp1
Cp2.Nom = "Henri"
```

Henri, 24
Henri, 24

Structure

Sp1

Objet



SPersonne sp1 ⇔ SPersonne sp1 = new SPersonne() car la valeur de sp1 est la structure elle-même. La valeur d'un objet est l'adresse de cet objet.

IX Les Interfaces

Une interface est un ensemble de prototypes de méthodes ou propriété qui forment un contrat. Une classe qui implémente cette interface s'engage à fournir toutes les méthodes de l'interface.

```
public interface IStats {
    double Moyenne { get; }
    double EcartType();
}
```

Pour code.

5) Les classes, interfaces et méthodes génériques

Supposons que l'on veuk echanger deux entiers:

```
public static void Echanger1 (ref int val1, ref int val2)
{
    int tmpval = val1;
    val1 = val2;
    val2 = tmpval;
}
```

Si on avait mis des Personne ou des double le code aurait été le même - Il n'y a que le type qui changeait.

```
class Generic1 <T> {
    public static void Echanger (ref T val1, ref T val2)
    {
        T tmpval = val1;          T variable muette
        val1 = val2;
        val2 = tmpval;
    }
```

Pour l'appel :

```
int i1=1, i2=2;
Generic1<int>.Echanger (ref i1, ref i2);
Personne p1 = new Personne ("E", "T", 36);
p2 ---------- ("toto", "tata", 24);
Generic1<Personne>.Echanger (ref p1, ref p2);
```

On aurait pu se rendre Generic que la méthode:

```
class Generic 2 {
    pub stat void Echanger <T> (ref T val1, ref T val2)
...
```

L'appel se fait par Generic2.Echanger <int> ...

On peut mettre des contraintes

```
class Generic1 <T> where T:
```

- struct
- class
- new() (l'argument doit avoir un constructeur ne prenant pas de paramètres)
- <base_class_name>
- <interface_name> (ou de ses dérivées)