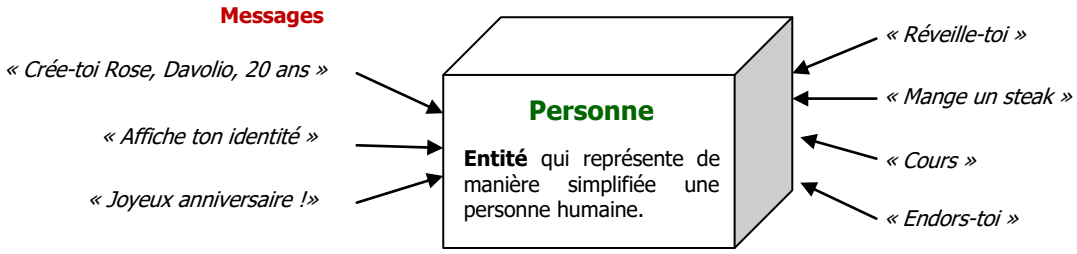
Initiation à la programmation orientée objet

I) Présentation

Pour représenter, de manière simplifiée, des PERSONNES, il est possible de concevoir une entité logicielle appelée « Personne » dont le fonctionnement revient à interpréter certains messages que l'on pourra lui envoyer :

- Une Personne peut être créée en recevant un message de demande d'initialisation, dans lequel on communique son PRENOM, NOM et AGE.
- Une Personne peut communiquer son identité en recevant un message de demande d'identification.
- Une Personne peut fêter son anniversaire en recevant le message « Joyeux anniversaire ».
- Une Personne peut, enfin, exercer une activité de la vie courante, telle que s'endormir, se réveiller, courir ou manger, en recevant le message correspondant.



Une telle entité, du point de vue de l'utilisateur, est une BOITE NOIRE qui réagit seulement à la réception de messages.

II) Objets, attributs et méthodes

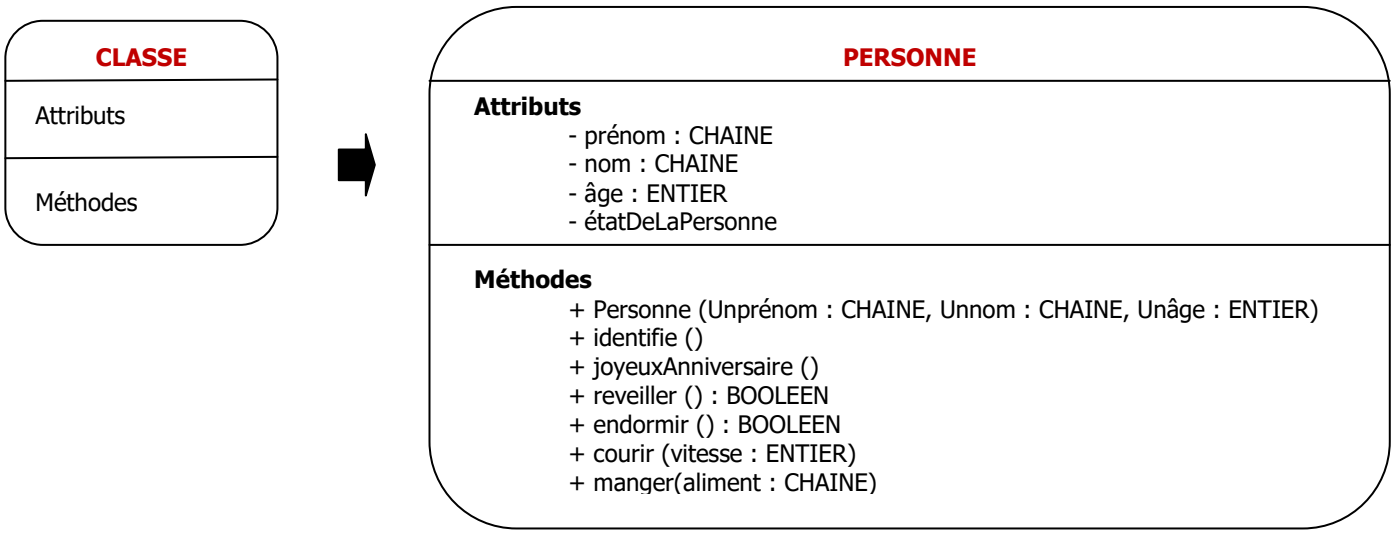
Un **objet** est une entité logicielle qui :

1. A une **IDENTITE** (son nom ou identificateur), tout comme une variable « classique » de type entier, chaîne, booléen…
2. Est capable de sauvegarder un **ETAT**, c'est-à-dire un ensemble d'informations (par exemple : le prénom, le nom, l'âge…) dans ses variables internes (ou **ATTRIBUTS**). Les états d'un objet sont les valeurs de ses variables. Par exemple :

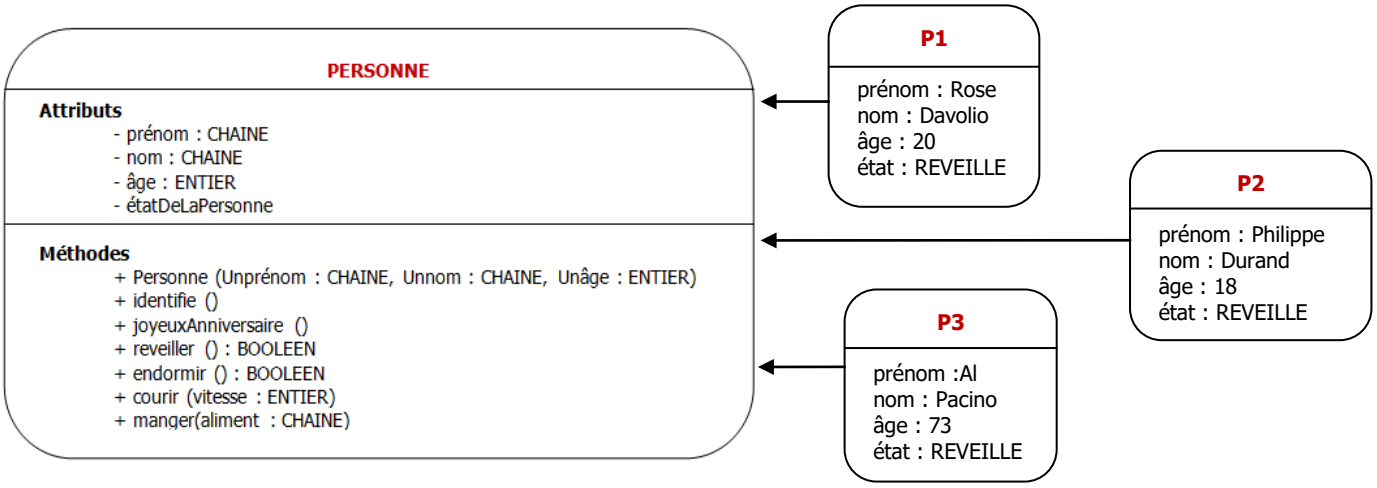
   VAR prénom, nom : CHAINE — prénom = « Rose » ; nom = « Davolio » ; âge = 20 ;
   âge : ENTIER
3. Répond à des messages précis en déclenchant des opérations internes qui changent l'état de l'objet (par exemple, le message « Joyeux anniversaire » va déclencher un traitement qui va ajouter un an à l'âge de la personne). Les messages auxquels l'objet peut répondre sont en fait les intitulés de ces opérations, qui sont en réalité des fonctions avec leurs paramètres et leur valeur de retour. En programmation objet, on les désigne plutôt avec le terme **METHODES**.

III) Classe et instances

Une classe est un ensemble d'objets qui ont en commun les mêmes méthodes et qui partagent les mêmes types d'attributs. Une classe est en fait un **TYPE DE DONNEES** qui regroupe à la fois des données (les attributs) et un comportement (les méthodes).



La classe es une sorte de MOULE qui permet ensuite de créer autant d'objets (ou INSTANCES) que l'on veut :



Chaque instance a un ETAT (valeurs des attributs) qui lui est propre. On pourra alors envoyer à chaque instance des messages qui **activeront les METHODES correspondantes**. Une instance se comporte en réalité comme une « super-variable » issue d'un « type de données évolué ».

Exemple : Le message « Joyeux anniversaire » envoyé à l'objet « P1 », déclenche l'exécution de la méthode « joyeuxAnniversaire() » qui va :

- Ajouter un an à la valeur de son attribut « age » : 20 +1 = 21 ans.
- Afficher un message : « 21 ans... Joyeux anniversaire Rose !

REMARQUES :

- **La liste des méthodes constitue l'INTERFACE de la classe**. Cela correspond aux différents SERVICES (ou opérations) disponibles qu'un objet ou instance pourra activer.
- **Les méthodes de tout objet opèrent sur ses attributs, qui sont en principe MASQUES** à l'environnement de l'objet (le programme utilisateur). On dit que les attributs restent PRIVES à l'objet (dans le schéma précédent, les attributs privés sont précédés par le signe « **-** ») : c'est le principe d'ENCAPSULATION des données.
- **Les membres (attributs ou méthodes) qui doivent être CONNUS** de l'environnement de l'objet, devront être déclarés PUBLICS (dans le schéma précédent, les méthodes publiques sont précédés par le signe « **+** »). L'accès à ces membres publics par un objet se fera par la notation suivante :

Nom de l'objet . identificateur du membre

Exemple : P1.identifie(), P1.joyeuxAnniversaire(), etc…

IV) Mise en œuvre en C#, sous Visual Studio. Notion de COMPOSANT logiciel.

Dans une première étape, il va s'agir de se placer du point de vue de l'UTILISATEUR pour développer une application informatique en utilisant les services d'un COMPOSANT logiciel.

Un « composant » peut être vu comme une entité logicielle regroupant une ou plusieurs classes. Un composant est, à priori, « thématique », c'est-à-dire qu'il a été conçu et développé pour un rôle bien particulier (gestion de personnes et d'une association de personnes, gestion des clients, accès à une base de données…).

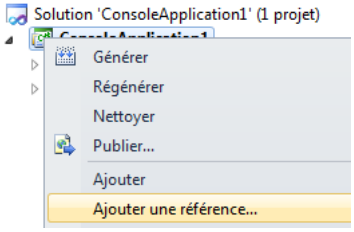
- Un composant regroupe un certain nombre de FONCTIONNALITES (des méthodes d'une classe) qui peuvent être appelées depuis un programme externe, ou client (à développer dans cette partie). Comme le composant ne contient que du code compilé, il n'est a priori pas possible de savoir de quelle manière elles sont implémentées, à moins de disposer du code source.
- De plus, pour pouvoir être utilisé, le composant doit fournir une INTERFACE, c'est-à-dire un ensemble de fonctions (ou méthodes) lui permettant de communiquer avec le programme client.
- Enfin, un composant doit être REUTILISABLE, c'est-à-dire qu'il ne doit pas simplement servir dans le cadre du projet durant lequel il a été développé.

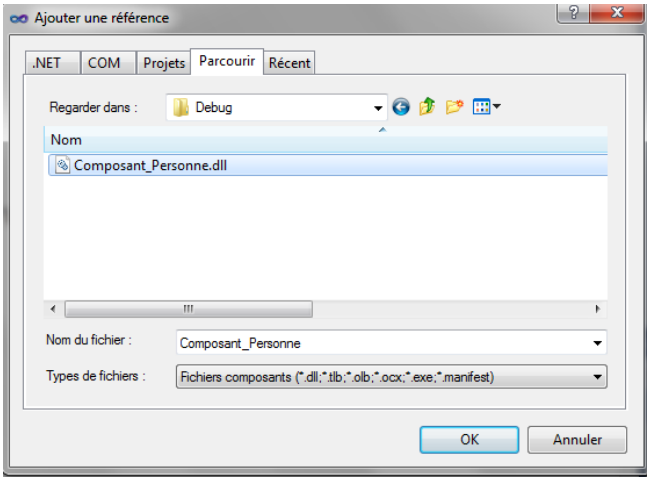
Lorsque l'on parle de composants, il s'agit de simples fichiers, contenant généralement du code compilé. Sous les systèmes de type Microsoft Windows, il s'agit des fameuses dll (« dynamic library link »). On parle également de modules, de bibliothèques, ou de librairies, par abus de traduction (« library » étant un faux ami et signifiant bibliothèque).Dans le Framework .NET, une « dll » est aussi appelée « assembly ».

Etape n°1

Créer d'abord un projet « Console ». Il s'agit ensuite d'ajouter une référence vers le composant (dll) fourni.

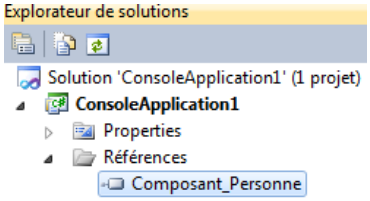
Pour cela, clic-droit sur l'icône du projet et choisir « Ajouter une référence »…



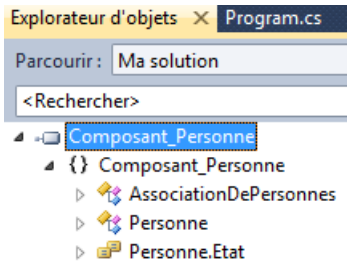


Le composant apparaît désormais comme une référence du projet (parmi les autres composants .NET).

Il suffit de double-cliquer sur l'icône pour afficher la fenêtre « Explorateur d'objets »



Fenêtre « Explorateur d'objets »

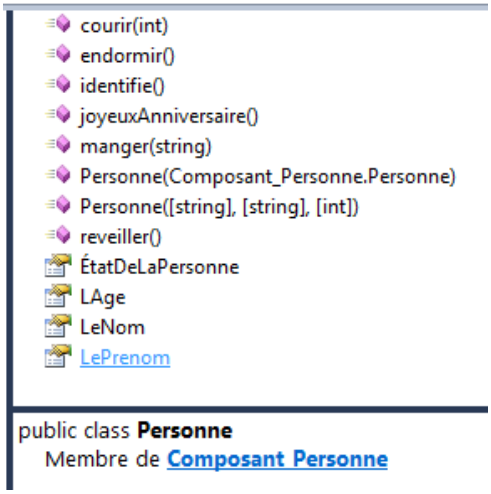


Le contenu du composant montre DEUX classes : « Personne » et « AssociationdePersonnes ». Une énumération « Personne.Etat » est disponible. Elle se comporte comme une liste de CONSTANTES qui correspondent aux différents états d'une personne :

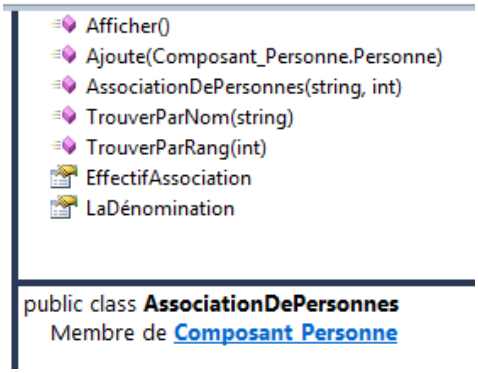
```
/// <summary>
/// Liste de constantes pour définir l'état d'une personne
/// </summary>
public enum Etat { REVEILLE, ENDORMI, FATIGUE, ENFORME };
```

Un clic sur chacune des classes, permet d'afficher sur la fenêtre de droite l'INTERFACE (membres publiques) correspondante :

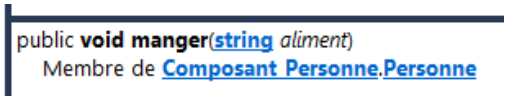
Classe « Personne »



Classe « AssociationDePersonnes »



Une méthode a sa propre SIGNATURE (nom, paramètres et type de données retourné). Il est indispensable de la connaître afin de pouvoir l'utiliser dans un programme client. Le clic sur une méthode affiche automatiquement sa signature, par exemple :



Etape n°2

Il s'agit maintenant de développer une application CLIENTE qui utilise les classes contenues dans le composant référencé. Pour cela :

- On va déclarer et créer des objets sur les classes.
- Chaque objet ainsi crée va pouvoir utiliser les services offerts (méthodes) par l'appel des méthodes correspondantes.

Architecture du projet à développer

```
// Lien vers des composants .NET
using System;

// Application CLIENTE qui utilise le composant "Composant_Personne",
// bibliothèque de classes {Client et AssociationDePersonnes}
namespace Personnes_Et_Association
{
    // Lien vers le composant pour accéder aux classes
    using Composant_Personne;

    /// <summary>
    /// Classe de TEST
    /// </summary>
    class Program
    {
        // Point d'entrée du projet ("programme principal")
        static void Main(string[] args)...
    }
}
```

Méthode « Main » de la classe de test « Program » : création d'une personne

```
//----------------------------
// TEST de la classe PERSONNE
//----------------------------

// Objet de classe Personne : déclaration + construction
Personne pacino = new Personne("Al", "Pacino", 73);

// Avant et après son anniversaire :
//  a. Affichage par identifie()
//  b. Joyeux anniversaire !
//  c. Affiche avec les propriétés
pacino.identifie();
pacino.joyeuxAnniversaire();
Console.WriteLine("Personne : {0} {1}, {2} ans", pacino.LePrenom, pacino.LeNom, pacino.LAge);
```

Instanciation d'un objet de classe « Personne »

a) Déclaration d'un objet de classe « Personne ».
b) Allocation en mémoire (« new ») de suffisamment de « place » pour gérer un objet « Personne ».
c) L'adresse de cet espace mémoire est stockée dans l'objet (appelé également en C# « référence »).
d) Lors de l'instanciation, les paramètres nécessaires pour « construire » l'objet (prénom, nom et l'âge) sont transmis.

Appel des méthodes (services ou fonctionnalités) à partir de l'objet crée.

Accès aux valeurs des attributs de l'objet

Des méthodes particulières, appelées « accesseurs » permettent d'accéder individuellement aux valeurs de chacun des attributs.

```
[Al,Pacino,73]
74 ans...Joyeux anniversaire Al !
Personne : Al Pacino, 74 ans
```

TRAVAIL A FAIRE

A partir de l'objet précédemment instancié, en utilisant les méthodes de la classe «Personne », compléter le code précédent afin d'obtenir le résultat suivant :

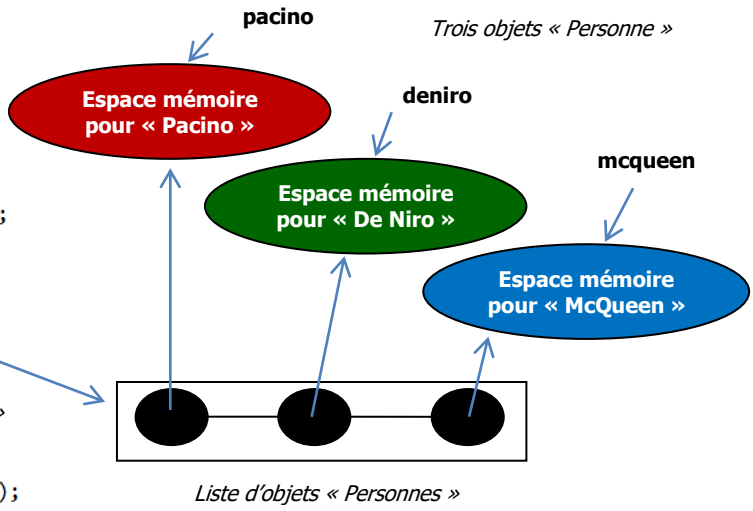
```
Tentative de mise à jour de l'âge
Erreur : Le format de la chaîne d'entrée est incorrect.
Age après maj : 68
Je cours à 5 km/h !
Je suis trop fatigué... il faut m'alimenter !
steak, c'est bon et merci la forme !
Je n'ai pas besoin d'aliment !
Impossible de m'endormir... Je suis trop en forme !
Vitesse de 50 km/h irréaliste !
Je cours à 5 km/h !
Dodo... ZZZZ !!!
Ne pas déranger !
Al réveille-toi !
Bonjour le monde !
```

En ce qui concerne les CHANGEMENTS D'ETAT que l'on peut appliquer à une personne, quatre méthodes sont disponibles qui respectent les règles suivantes :

- **reveiller()** : on ne peut pas réveiller une personne déjà réveillée, et dans tous les cas, la personne doit être endormie !
- **endormir()** : on ne peut pas endormir une personne si elle est déjà endormie ou si elle est trop en forme (il faut alors la fatiguer !).
- **courir(vitesse)** : pour faire courir une personne, elle ne doit pas être endormie (il faut donc la réveiller) ou trop fatiguée (il faut alors la faire manger !). Par contre la vitesse humaine ne dépasse pas 40 km/h !
- **manger(aliment)** : pour nourrir une personne il faut la réveiller auparavant. Mais il ne faut abuser ; on ne peut nourrir une personne qui a déjà mangé.

Méthode « Main » de la classe de test « Program » : création d'une association de Personnes

```
//-----------------------------------------
// TEST de la classe ASSOCIATION DE PERSONNES
//-----------------------------------------

// Créer une association
AssociationDePersonnes acteurs = new AssociationDePersonnes("Association d'acteurs", 3);

// Créer deux personnes
Personne deniro = new Personne("Robert", "De Niro", 68);
Personne mcqueen = new Personne("Steve", "McQueen", 50);

// Avant l'ajout, tous les acteurs sont réveillés...
// On endort un parmi eux...
if (deniro.endormir())
    Console.WriteLine("Un des acteurs vient de s'endormir");

// Ajout de la personne créée dans la première partir,
// puis les deux précédentes...
acteurs.Ajoute(pacino);
acteurs.Ajoute(deniro);
acteurs.Ajoute(mcqueen);

// Nb max atteint !
Console.WriteLine();
acteurs.Ajoute(new Personne("Jean-Claude", "Van Damme", 45));
```




```
Dodo... ZZZZ !!!
Un des acteurs vient de s'endormir

Nombre max de participants atteint; ajout de Van Damme impossible !
```

TRAVAIL A FAIRE

A partir de l'objet précédemment instancié, en utilisant les méthodes de la classe « AssociationDePersonnes », compléter le code précédent afin d'obtenir le résultat suivant :

```
Liste des membres :
[Al,Pacino,68]
[Robert,De Niro,68]
[Steve,McQueen,50]

Saisir le nom d'un membre : Gimenez
Membre non trouve

Il y a au moins un membre endormi !
```

```
Liste des membres :
[Al,Pacino,68]
[Robert,De Niro,68]
[Steve,McQueen,50]

Saisir le nom d'un membre : McQueen
Résultat : [Steve,McQueen,50]

Il y a au moins un membre endormi !
```

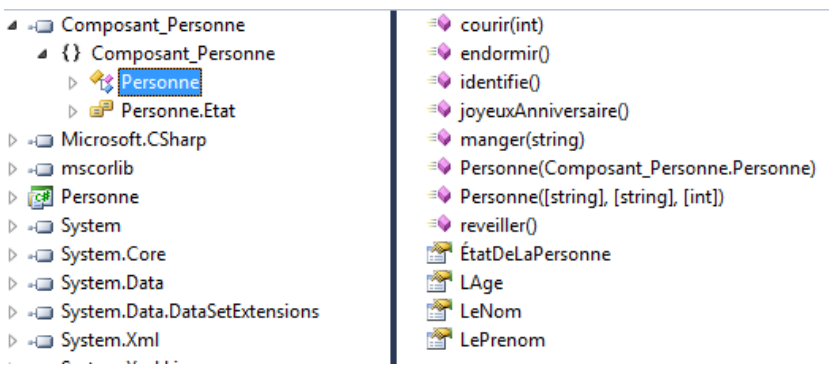
V) Utilisation d'un composant et développement d'une classe des gestion d'une liste d'objets

Il s'agira maintenant d'adopter une approche mixte :

- **Se placer du point de vue de l'utilisateur d'un composant logiciel contenant la seule classe « Personne ».**
- **Puis, se placer du point de vue du concepteur et créateur de sa propre classe, qui va permettre de gérer une liste d'objets « Personnes ».**

Etape n°1 : développer une application de gestion d'une liste de « personnes ». (utilisation de TABLEAUX).

Créer un nouveau projet, puis ajouter une référence vers un nouveau composant :



Architecture du projet à développer

```
using System;

namespace ListeDePersonnes
{
    // Lien vers le composant où se trouve la classe Personne
    using Composant_Personne;

    /// <summary>
    /// Classe de TEST
    /// </summary>
    class Program
    {
        static void Main(string[] args)...
    }
}
```

Méthode « Main » de la classe de test « Program » : gestion d'une association de Personnes

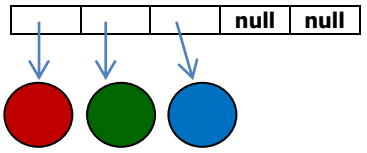
```
// Déclaration, puis allocation mémoire
Personne[] liste = new Personne[5];

// Initialisation avec la valeur "null"
for (int i=0; i<liste.Length; i++)
    liste[i] = null;
```

Structure de données

Un TABLEAU de « Personnes » servira à stocker des objets de classe « Personne ». Une taille doit être spécifiée (ici 5). Le tableau est initialisé avec la valeur « nul », c'est-à-dire que chaque élément du tableau ne référence aucun objet « Personne ».

```
// Remplissage

// a. Créer des objets Personne
Personne p1 = new Personne("Al", "Pacino", 73);
Personne p2 = new Personne("Steve", "McQueen", 50);
Personne p3 = new Personne("Robert", "De Niro", 69);

// b. Puis stocker leurs adresses dans le tableau
liste[0] = p1;
liste[1] = p2;
liste[2] = p3;
```

TABLEAU liste




```
// Parcours de la liste, récupération de chaque personne,
// puis accès aux méthodes de la classe Personne
Personne unePersonne;
for (int i = 0; liste[i] != null; i++)
{
    unePersonne = liste[i];
    unePersonne.identifie();
}

Console.WriteLine();
```

Extraction d'un élément

Chaque élément de rang « i » retourne une référence sur la classe « Personne ». L'objet ainsi obtenu est capable d'appeler les méthodes de sa classe.

```
// Ajout de 2 personnes avec création "directe"
liste[3] = new Personne("Paul", "Newman", 65);
liste[4] = new Personne("Robert", "Redford", 65);

// Un ajout supplémentaire povoquerait une erreur
// car le tableau est plein !

// a. Compter le nombre d'éléments
int nb=0;
for (int i = 0; i < liste.Length; i++)
{
    if (liste[i] != null)
        nb++;
}

// b. Test avant l'ajout
if (nb == liste.Length)
    Console.WriteLine("Nombre MAX de personnes atteint !");
else
    liste[4] = new Personne("Jean-Claude", "Van Damme", 50);


// Parcours de la liste et affichage
for (int i = 0; i < liste.Length; i++)
{
    if (liste[i] != null)
        liste[i].identifie();
}

Console.WriteLine();
Console.WriteLine();
```

Extraction d'un élément (bis)

L'élément de rang « i » est déjà un objet de classe « Personne ». On peut dont appeler directement la méthode « identifie() » en utilisant cette notation simplifiée.

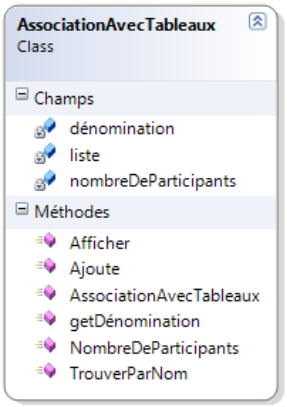
```
        // Recherche d'une personne dans la liste par son nom
        Console.Write("Nom de la personne à rechercher : ");
        string nom = Console.ReadLine();
        if (nom.Length != 0)    // Si quelque chose a été saisi
        {
            int i=0;
            bool trouve, fini;
            trouve = fini = false;
            while (!trouve && !fini)
            {
                if (liste[i].LeNom == nom)
                    trouve = true;
                else
                {
                    i++;    // Passage à l'élément suivant
                    if (i == liste.Length)    // Tous les élements ont été parcourus
                        fini = true;
                }
            }

        Console.WriteLine();

        // Bilan de la recherche
        if (trouve)
        {
            Console.WriteLine("TROUVE ! ");
            liste[i].identifie(); // La valeur du compteur 'i' permet d'accéder à la personne recherchée
        }
        else
            Console.WriteLine("{0} NON TROUVE ! ", nom);
    }
```

Utilisation d'un « accesseur » sur le nom

L'élément de rang « i » permet d'utilisation d'un accesseur « LeNom » qui va récupérer la valeur de l'attribut « nom « de l'objet de rang « i ». Cette valeur est comparée avec le nom précédemment saisi.

TRAVAIL A FAIRE

En vous inspirant du code précédent, écrire une CLASSE de gestion d'une liste d'objets « Personne », en utilisant un TABLEAU. On vous fournit la déclaration suivante de la classe :




```
/// <summary>
/// Classe ASSOCIATION de PERSONNES
/// -- Utilisation de tableaux --
/// </summary>
public class AssociationAvecTableaux
{
    // Attributs
    string dénomination;        // Dénomination de l'association
    Personne[] liste;           // Liste de Personnes
    int nombreDeParticipants;   // Nb courant de participants pour le tableau

    // Constructeur (on lui passe la dénomination et le nombre MAX de participants)
    public AssociationAvecTableaux(string d, int max)...

    //---------
    // Méthodes
    //---------

    // Ajoute une Personne à l'association
    public void Ajoute(Personne p)...

    // Renvoie une Personne après une recherche par le nom.
    // Si non trouvée, on retourne "null"
    public Personne TrouverParNom(string n)...

    // Affiche la liste des membres
    public void Afficher()...

    // Nombre de participants
    public int NombreDeParticipants()...

    //----------
    // Accesseur
    //----------
    public string getDénomination()...
}
```

Etape n°2 : développer une application de gestion d'une liste de « personnes ». Utilisation de COLLECTIONS

Les collections d'objets sont des CLASSES du Framework .NET qui permettent de stocker et de manipuler une liste d'objets. Une collection est équivalant à un tableau mais avec une des fonctionnalités (méthodes) qui rendant plus aisée la manipulation de la liste d'objets.

Exemple :

```
// Pour travailler avec une collection "List"
using System.Collections.Generic;
```

```
// Collection d'objets de CLASSE "PERSONNE"
List<Personne> LesEmployes = new List<Personne>();
```

Structure de données

Un objet de classe COLLECTION (« List ») va stocker une liste d'objets de classe « Personne ». Contrairement aux tableaux, la taille n'a pas besoin d'être spécifiée. En réalité, la collection se comporte comme un tableau DYNAMIQUE, c'est-à-dire qu'au départ la liste est VIDE, puis sa taille va augmenter « dynamiquement » à chaque ajout d'un objet « Personne ».

```
// 1. Ajout d'objets à la collection
Personne p1 = new Personne("Jean", "Dupont", 30);
Personne p2 = new Personne("Marie", "Duval", 25);
Personne p3 = new Personne("Philippe", "Durand", 21);
LesEmployes.Add(p1);
LesEmployes.Add(p2);
LesEmployes.Add(p3);
```

Ajout à la liste

La méthode « Add() » de la classe « List » permet d'ajouter un élément à la liste. Cet élément va s'ajouter à la fin.

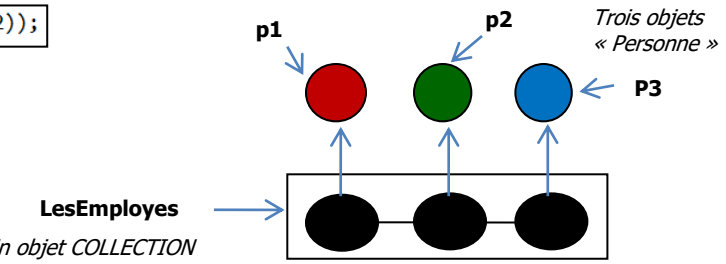
```
// 2. Nombre d'objets de la collection
Console.WriteLine("Nombre d'employés : " + LesEmployes.Count);
Console.WriteLine();
```

Nombre d'éléments de la liste

La propriété « Count » fournit en temps réel le nombre d'objets dans la liste.

```
// 3. Recherche de l'INDEX d'un objet dans la collection
p2.identifie();
Console.Write(" a un index = " + LesEmployes.IndexOf(p2));
Console.WriteLine();
```

```
// 4. Recherche si un objet EXISTE dans la collection
if (LesEmployes.Contains(p2))
{
    p2.identifie();
    Console.Write(" existe dans la collection");
}
else
{
    p2.identifie();
    Console.Write(" n'existe pas dans la collection");
}
```



Récupération d'un élément de la liste

Pour récupérer un élément de la liste on utilise la notation TABLEAU :

objet collection[rang de l'élément].

Ici, la méthode « IndexOf() » donne le rang de l'objet « p3 », soit 2. Cette valeur est ensuite utilisée pour accéder à cet élément.

```
// 5. RECUPERATION d'un objet de la collection
Personne q = LesEmployes[LesEmployes.IndexOf(p3)];
Console.WriteLine();
Console.WriteLine();
q.identifie();
Console.Write(" a été récupéré de la collection ");
```

```
// 6. AJOUT d'un objet à la collection
LesEmployes.Insert(LesEmployes.Count, new Personne("Paul", "Dumond", 50));
Console.WriteLine();
Console.WriteLine();
Console.WriteLine("Nombre d'employés après ajout : " + LesEmployes.Count);
```

```
// 7. SUPPRESSION d'un objet de la collection
LesEmployes.Remove(p1);
Console.WriteLine();
Console.WriteLine("Nombre d'employés après suppression : " + LesEmployes.Count);
Console.WriteLine();
```

```
// 8. PARCOURS de la collection : (cf. TABLEAU)
for (int i = 0; i < LesEmployes.Count; i++ )
{
    Console.Write("Employé n° " + i + " ");
    LesEmployes[i].identifie();
    Console.WriteLine();
}
```

Autres fonctionnalités

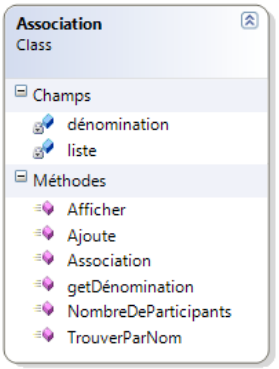
- **Insert()** permet d'ajouter un élément à un emplacement donnée.
- **Remove()** supprime un objet de la liste.
- **LesEmployes[i]** fournit un objet de classe « Personne ». On peut alors utiliser les méthodes disponibles de la classe « Personne ».
- **Clear()** permet de vider la collection.

```
// 8. VIDER la collection
LesEmployes.Clear();
Console.WriteLine();
Console.WriteLine("Après suppression de la collection : " + LesEmployes.Count);
```

```
Nombre d'employés : 3

Marie,Duval,25 a un index = 1
Marie,Duval,25 existe dans la collection

Philippe,Durand,21 a été récupéré de la collection

Nombre d'employés après ajout : 4

Nombre d'employés après suppression : 3

Employé n° 0 Marie,Duval,25
Employé n° 1 Philippe,Durand,21
Employé n° 2 Paul,Dumond,50

Après suppression de la collection : 0
```

TRAVAIL A FAIRE

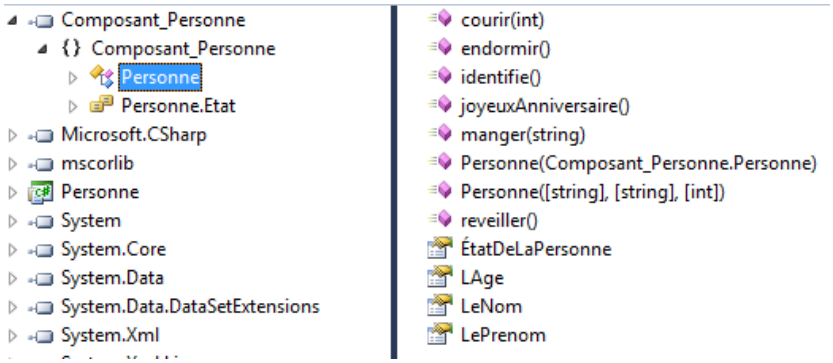
En vous inspirant du code précédent, écrire une CLASSE de gestion d'une liste d'objets « Personne », en utilisant une COLLECTION. On pourra écrire cette classe en adaptant la classe « AssociationAvecTableaux » précédemment écrite. On vous fournit la déclaration suivante de la classe :




```
/// <summary>
/// Classe ASSOCIATION de PERSONNES
/// -- Utilisation de collection --
/// </summary>
public class Association
{
    // Attributs
    string dénomination;    // Dénomination de l'association
    ArrayList liste;        // Liste de Personnes

    // Constructeur (on lui passe la dénomination)
    public Association(string d)...

    //---------
    // Méthodes
    //---------

    // Ajoute une Personne à l'association
    public void Ajoute(Personne p)...

    // Renvoie une Personne après une recherche par le nom.
    // Si non trouvée, on retourne "null"
    public Personne TrouverParNom(string n)...

    // Affiche la liste des membres
    public void Afficher()...

    // Nombre de participants
    public int NombreDeParticipants()...

    //----------
    // Accesseur
    //----------
    public string getDénomination()...
}
```

VI) Introduction à la notion d'HERITAGE et de POLYMORPHISME

On dispose du composant précédent, qui met à notre disposition la classe « Personne » avec les différentes méthodes :



On souhaite maintenant développer et utiliser deux classes : « Enseignant » et « Etudiant ». Les objets issus de ces classes sont très proches des objets de la classe « Personne » :

- Ils ont en COMMUN un prénom, un nom et un âge, puis toutes les fonctionnalités définies dans la classe « Personne », telles que « identifie() », « joyeuxAnniversaire() », etc…
- Cependant, pour un enseignant on souhaite gérer en plus la matière enseignée, et donc obtenir un affichage spécifique, et pour un étudiant, il s'agira de gérer la classe à laquelle il appartient.

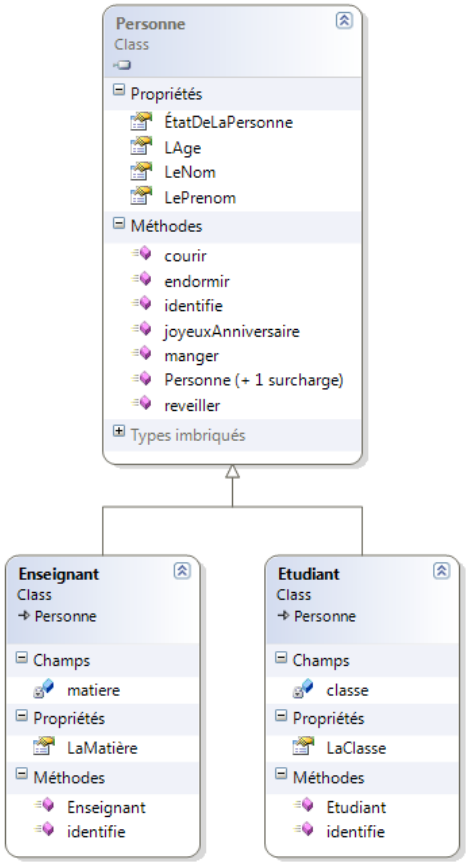
Les options suivantes sont offertes au développeur :

1) **Dupliquer le code de la classe « Personne », puis l'adapter (ou compléter).** Ceci pose le problème de la redondance du code (la modification ultérieure d'une méthode de « personne » devra être répercutée sur toutes ses copies dans les classes « Enseignant » et « Etudiant ». De plus, si ce code est INACCESSIBLE, comme c'est le cas ici (le composant utilisé contient du code compilé), cette option devient impossible !

2) **Réécrire la totalité du code.** Ceci résout le problème d'e l'inaccessibilité du code de la classe « Personne », mais contraint à passer du temps à développer des fonctionnalités déjà écrites !

3) **Créer les deux classes « Enseignant » et « Etudiant » par DERIVATION de la classe « Personne ».** On dit alors que ces deux classes vont HERITER des attributs et méthodes de la classe « Personne » (qui devient la classe « mère »). Les classes dérivées (ou « filles ») pourront alors implémenter des méthodes spécifiques supplémentaires et utilisant (ou non) des fonctionnalités héritées.

C'est cette dernière possibilité qui est retenue ici :




```
// Classe ENSEIGNANT (fille) qui derive de PERSONNE
public class Enseignant : Personne
{
    // Attribut privé SPECIFIQUE : matière enseignée
    private string matiere;

    // Propriété publique sur l'attribut
    public string LaMatière
    {
        get { return matiere; }
        set { matiere = value; }
    }

    // Constructeur qui doit appeller celui de la classe MERE
    public Enseignant(string p, string n, int a, string m): base(p, n, a)
    {
        Console.WriteLine("Construction enseignant");
        matiere = m;
    }

    // Méthode héritée "REDEFINIE"
    public override void identifie()
    {
        Console.Write("Enseignant : ");

        // Appel méthode HERITEE
        base.identifie();

        // Personnalisation de l'affichage
        Console.WriteLine("enseigne : " + matiere);
    }
}
```

Lien d'HERITAGE

Attribut SPECIFIQUE

Les autres (prénom, nom et âge sont hérités).

Construction de la partie « Personne »

Pour construire un objet « Enseignant », on doit communiquer à son constructeur : son prénom, nom âge, puis la matière enseignée.

Le constructeur doit d'abord appeler le constructeur de « Personne » en lui transmettant les paramètres requis avant d'initialiser sont attribut spécifique.

Redéfinir (personnaliser) une méthode héritée

La classe « Enseignant » hérite des méthodes publiques de « Personne ». Dans ce cas précis, on souhaite PERSONNALISER la méthode « identifie() ».
Au lieu de la REECRIRE en totalité, on va appeler la méthode héritée : **base.identifie()**, puis ajouter du code supplémentaire.

TRAVAIL A FAIRE

En vous inspirant du code précédent, écrire le code correspondant à la classe « Etudiant ». Un étudiant aura, en plus d'un prénom, nom et âge, et des fonctionnalités de la classe « Personne », un attribut spécifique « nom de sa classe » de type chaîne de caractères.

On souhaite par ailleurs, redéfinir la méthode héritée « identifie() » afin d'en personnaliser l'affichage (inclure lors de l'affichage le nom de sa classe).

Exemple d'utilisation avec mise en œuvre de l'héritage et introduction à la notion de polymorphisme :

```
//------------------------
// Une personne "standard"
//------------------------

Personne p = new Personne("Rose", "Davolio", 25);
// Affichage à travers les propriétés, puis la méthode "identifie()"
Console.WriteLine("PERSONNE => prenom : {0} nom : {1} age : {2}", p.LePrenom, p.LeNom, p.LAge);
p.identifie();

Console.WriteLine();
```

```
PERSONNE => prenom : Rose nom : Davolio age : 25
[Rose,Davolio,25]
```

```
//-------------------------------------------
// La personne est maintenant un enseignant...
//-------------------------------------------

// a. Construction (éléments pour Personne, puis pour Enseignant)
p = new Enseignant("Jean", "Dupont", 30, "Informatique");

// b. Membres HERITES
Console.WriteLine("ENSEIGNANT => prenom : {0} nom : {1} age : {2}", p.LePrenom, p.LeNom, p.LAge);
p.joyeuxAnniversaire();
p.manger("hamburger");
Console.WriteLine("Etat : " + p.ÉtatDeLaPersonne);

// c. POLYMORPHISME : la méthode REDEFINIE est appelée automatiquement
p.identifie();

Console.WriteLine();
```

```
Construction enseignant
ENSEIGNANT => prenom : Jean nom : Dupont age : 30
31 ans...Joyeux anniversaire Jean !
hamburger, c'est bon et merci la forme !
Etat : ENFORME
Enseignant : [Jean,Dupont,31]
enseigne : Informatique
```

```
//-------------------------------------------
// La personne est maintenant un étudiant...
//-------------------------------------------

// a. Construction (éléments pour Personne, puis pour Etudiant)
p = new Etudiant("Paul", "Dumoulin", 18, "S1SIO");

// b. Membres HERITES
Console.WriteLine("ETUDIANT => prenom : {0} nom : {1} age : {2}", p.LePrenom, p.LeNom, p.LAge);
p.joyeuxAnniversaire();
p.manger("hamburger");
p.courir(10);
if (p.endormir())
    Console.WriteLine("Ne pas déranger...");

Console.WriteLine("Etat : " + p.ÉtatDeLaPersonne);

// c. POLYMORPHISME : la méthode REDEFINIE est appelée automatiquement
p.identifie();
```

```
Construction étudiant
ETUDIANT => prenom : Paul nom : Dumoulin age : 18
19 ans...Joyeux anniversaire Paul !
hamburger, c'est bon et merci la forme !
Je cours à 10 km/h !
Dodo... ZZZZ !!!
Ne pas déranger...
Etat : ENDORMI
Etudiant : [Paul,Dumoulin,19]
sa classe : S1SIO
```