Chapitre 6 : Programmation Orientée Objets

6.1 Définition d'une classe

6.1.1 Définition de la classe Personne

La définition de la classe *Personne* dans le fichier source [*Personne.cs*] sera la suivante :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Chapitre6
{
    class Personne
    {
        // attributs
        private string prenom;
        private string nom;
        private int age;

        // méthode
        public void Initialise(string P, string N, int age)
        {
            this.prenom = P;
            this.nom = N;
            this.age = age;
        }

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

        }
    }
```

Une classe → type de données.
Les objets → instances de classe.

Les attributs peuvent être accompagnés de l'un des trois mots clés suivants :

- privé : Un champ privé (private) n'est accessible que par les seules méthodes internes de la classe
- public : Un champ public (public) est accessible par toute méthode définie ou non au sein de la classe
- protégé : Un champ protégé (protected) n'est accessible que par les seules méthodes internes de la classe ou d'un objet dérivé.

6.1.2 La méthode Initialise

Revenons à notre classe **Personne.**

Quel est le rôle de la méthode I*nitialise* ?

Parce que *nom*, *prenom* et *age* sont des données **privées** de la classe P*ersonne*, les instructions :

```
Personne p1;
p1.prenom = "Mohamed";
p1.nom = "Salah";
p1.age = 30;
```

sont illégales.

Il nous faut initialiser un objet de type *Personne* via une méthode publique. C'est le rôle de la méthode *Initialise*. On écrira :

```
Personne p1;
p1.Initialise("Mohamed", "Salah", 30);
```

6.1.3 L'opérateur new

La séquence d'instructions

```
Personne p1;
p1.Initialise("Mohamed", "Salah", 30);
```

est incorrecte.
→ L'instruction *Personne p1;* déclare *p1* comme une référence à un objet de type *Personne*. Cet objet n'existe pas encore et donc *p1* n'est pas initialisé.

Lorsqu'on écrit ensuite *p1.Initialise("Mohamed","Salah",30);*
on fait appel à la méthode *Initialise* de l'objet référencé par *p1*. Or cet objet n'existe pas encore et le compilateur signalera l'erreur.

Pour que *p1* référence un objet, il faut écrire :

```
Personne p1 = new Personne();
```

Cela a pour effet de créer un objet de type *Personne* non encore initialisé : les attributs *nom* et *prenom* qui sont des références d'objets de type *String* auront la valeur *null*, et *age* la valeur *0*. Il y a donc une initialisation par défaut. Maintenant que *p1* référence un objet, l'instruction d'initialisation de cet objet *p1.Initialise("Mohamed","Salah",30);* est valide.

6.1.4 Le mot clé this

L'instruction *this.prenom=p* signifie que l'attribut *prenom* de l'objet courant (*this*) reçoit la valeur *p*.
Le mot clé *this* désigne l'objet courant : celui dans lequel se trouve la méthode exécutée.
Lorsqu'une méthode d'un objet référence un attribut *A* de cet objet, l'écriture *this.A* est implicite. On doit l'utiliser explicitement lorsqu'il y a conflit d'identificateurs. C'est le cas de l'instruction :

this.*age=age;*

où *age* désigne un attribut de l'objet courant ainsi que le paramètre *age* reçu par la méthode. Il faut alors lever l'ambiguïté en désignant l'attribut *age* par *this.age*.

6.1.5 Un programme de test

Voici un court programme de test. Celui-ci est écrit dans le fichier source [*Program.cs*] :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Chapitre6
{
	class Program
	{
		static void Main(string[] args)
		{
			Personne p1 = new Personne();
			p1.Initialise("Mohamed", "Salah", 30);
			p1.Identifie();
		}
	}
}
```

6.1.6 Une autre méthode Initialise

Considérons toujours la classe *Personne* et rajoutons-lui la méthode suivante :

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

On a maintenant deux méthodes portant le nom Initialise : c'est légal tant qu'elles admettent des paramètres différents. C'est le cas ici. Le paramètre est maintenant une référence p à une personne. Les attributs de la personne p sont alors affectés à l'objet courant (this). On remarquera que la méthode Initialise a un accès direct aux attributs de l'objet p bien que ceux-ci soient de type private. C'est toujours vrai : un objet o d'une classe C a toujours accès aux attributs des objets de la même classe C.

Voici un test de la nouvelle classe Personne :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne p1 = new Personne();
            p1.Initialise("Mohamed", "Salah", 30);
            p1.Identifie();
            Personne p2 = new Personne();
            p2.Initialise(p1);
            p2.Identifie();
```

```
        }
    }
}
```

et ses résultats :

[Mohamed, Salah, 30]
[Mohamed, Salah, 30]

6.1.7 Constructeurs de la classe Personne

Un constructeur est une méthode qui porte le nom de la classe et qui est appelée lors de la création de l'objet. On s'en sert généralement pour l'initialiser. C'est une méthode qui peut accepter des arguments mais qui ne rend aucun résultat. Son prototype ou sa définition ne sont précédés d'aucun type (pas même *void*).
Si une classe C a un constructeur acceptant n arguments *argi*, la déclaration et l'initialisation d'un objet de cette classe pourra se faire sous la forme :

C objet =new C(arg1,arg2, ... argn);
ou
C objet;
…
objet=new C(arg1,arg2, ... argn);

Lorsqu'une classe *C* a un ou plusieurs constructeurs, l'un de ces constructeurs doit être obligatoirement utilisé pour créer un objet de cette classe.

Si une classe *C* n'a aucun constructeur, elle en a un par défaut qui est le constructeur sans paramètres : *public C()*. Les attributs de l'objet sont alors initialisés avec des valeurs par défaut. C'est ce qui s'est passé lorsque dans les programmes précédents, où on avait écrit :

```
Personne p1;
p1 = new Personne();
```

Créons deux constructeurs à notre classe *Personne* :

```
using System;

namespace Chapitre6
{
    class Personne
    {
        // attributs
        private string prenom;
        private string nom;
        private int age;
        // constructeurs
        public Personne(String p, String n, int age)
        {
            Initialise(p, n, age);
        }
        public Personne(Personne P)
        {
            Initialise(P);
        }

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

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

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

Nos deux constructeurs se contentent de faire appel aux méthodes *Initialise* étudiées précédemment. On rappelle que lorsque dans un constructeur, on trouve la notation *Initialise(p)* par exemple, le compilateur traduit par *this.Initialise(p)*. Dans le constructeur, la méthode *Initialise* est donc appelée pour travailler sur l'objet référencé par *this*, c'est à dire l'objet courant, celui qui est en cours de construction.
Voici un court programme de test :

```
using System;
namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne p1 = new Personne("Mohamed", "Salah", 30);
            p1.Identifie();
            Personne p2 = new Personne(p1);
            p2.Identifie();
        }
    }
}
```

et les résultats obtenus :

[Mohamed, Salah, 30]
[Mohamed, Salah, 30]

6.1.8 Les objets temporaires

Dans une expression, on peut faire appel explicitement au constructeur d'un objet : celui-ci est construit, mais nous n'y avons pas accès (pour le modifier par exemple). Cet objet temporaire est construit pour les besoins d'évaluation de l'expression puis abandonné. L'espace mémoire qu'il occupait sera automatiquement récupéré ultérieurement par un programme appelé "ramassemiettes" dont le rôle est de récupérer l'espace mémoire occupé par des objets qui ne sont plus référencés par des données du programme.

Considérons le nouveau programme de test suivant :

```
using System;

namespace Chapitre6
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            new Personne(new Personne("Mohamed", "Salah", 30)).Identifie();
        }
    }
}
```

et modifions les constructeurs de la classe *Personne* afin qu'ils affichent un message :

```
// constructeurs
public Personne(String p, String n, int age)
{
      Console.WriteLine("Constructeur Personne(string, string, int)");
      Initialise(p, n, age);
}
public Personne(Personne P)
{
      Console.Out.WriteLine("Constructeur Personne(Personne)");
      Initialise(P);
}
```

Nous obtenons les résultats suivants :

Constructeur Personne(string, string, int)
Constructeur Personne(Personne)
[Mohamed, Salah, 30]

montrant la construction successive des deux objets temporaires.

6.1.9 Méthodes de lecture et d'écriture des attributs privés

Nous rajoutons à la classe *Personne* les méthodes nécessaires pour lire ou modifier l'état des attributs des objets :

```
using System;

namespace Chapitre6
{
    class Personne
    {
        // attributs
        private string prenom;
        private string nom;
        private int age;

        // constructeurs
        public Personne(String p, String n, int age)
        {
            Console.WriteLine("Constructeur Personne(string, string,
            int)");
            Initialise(p, n, age);
        }
        public Personne(Personne p)
        {
            Console.Out.WriteLine("Constructeur Personne(Personne)");
            Initialise(p);
        }
        // méthode
        public void Initialise(string p, string n, int age)
        {
            this.prenom = p;
            this.nom = n;
            this.age = age;
```

```
        }
        public void Initialise(Personne p)
        {
            prenom = p.prenom;
            nom = p.nom;
            age = p.age;
        }
        // accesseurs
        public String GetPrenom()
        {
            return prenom;
        }
        public String GetNom()
        {
            return nom;
        }
        public int GetAge()
        {
            return age;
        }
        //modifieurs
        public void SetPrenom(String P)
        {
            this.prenom = P;
        }
        public void SetNom(String N)
        {
            this.nom = N;
        }
        public void SetAge(int age)
        {
            this.age = age;
        }
        // méthode
        public void Identifie()
        {
            Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
        }
    }
}
```

Nous testons la nouvelle classe avec le programme suivant :

```
using System;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne p = new Personne("Mohamed", "Mustapha", 34);
            Console.Out.WriteLine("p=(" + p.GetPrenom() + "," + p.GetNom()
            + "," + p.GetAge() + ")");
            p.SetAge(56);
            Console.Out.WriteLine("p=(" + p.GetPrenom() + "," + p.GetNom()
            + "," + p.GetAge() + ")");
        }
    }
}
```

et nous obtenons les résultats :

Constructeur Personne(string, string, int)

p=(Mohamed,Mustapha,34)
p=(Mohamed,Mustapha,56)

6.1.10 Les propriétés

Il existe une autre façon d'avoir accès aux attributs d'une classe, c'est de créer des propriétés. Celles-ci nous permettent de manipuler des attributs privés comme s'ils étaient publics.

Considérons la classe P*ersonne* suivante où les accesseurs et modifieurs précédents ont été remplacés par des **propriétés** en lecture et écriture :

```
using System;

namespace Chapitre6
{
    class Personne
    {   // attributs
        private string prenom;
        private string nom;
        private int age;
        // constructeurs
        public Personne(String p, String n, int age)
        {
            Initialise(p, n, age);
        }
        public Personne(Personne p)
        {
            Initialise(p);
        }
        // méthode
        public void Initialise(string p, string n, int age)
        {
            this.prenom = p;
            this.nom = n;
            this.age = age;
        }
        public void Initialise(Personne p)
        {
            prenom = p.prenom;
            nom = p.nom;
            age = p.age;
        }
        // propriétés
        public string Prenom
        {
            get { return prenom; }
            set
            {
                // prénom valide ?
                if (value == null || value.Trim().Length == 0)
                {
                    throw new Exception("prénom (" + value + ") invalide");
                }
                else
                {
                    prenom = value;
                }
            }//if
        }//prenom
```

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

        public int Age
        {
            get { return age; }
            set
            {
                // age valide ?
                if (value >= 0)
                {
                    age = value;
                }
                else
                    throw new Exception("âge (" + value + ") invalide");
            }//if
        }//age
        // méthode
        public void Identifie()
        {
            Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
        }
    }
}
```

Une propriété permet de lire (**get**) ou de fixer (**set**) la valeur d'un attribut. Une propriété est déclarée comme suit :

public Type Propriété
{
get {...}
set {...}
}

où *Type* doit être le type de l'attribut géré par la propriété. Elle peut avoir deux méthodes appelées **get** et **set**.

La méthode **get** est habituellement chargée de rendre la valeur de l'attribut qu'elle gère (elle pourrait rendre autre chose).

La méthode **set** reçoit un paramètre appelé **value** qu'elle affecte normalement à l'attribut qu'elle gère.

Elle peut en profiter pour faire des vérifications sur la validité de la valeur reçue et éventuellement lancer une exception si la valeur se révèle invalide. C'est ce qui est fait ici.

Comment ces méthodes **get** et **set** sont-elles appelées ? Considérons le programme de test suivant :

```
using System;

namespace Chapitre6
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne p = new Personne("Mohamed", "Salah", 34);
            Console.Out.WriteLine("p=(" + p.Prenom + "," + p.Nom + "," +
            p.Age + ")");
            p.Age = 56;
            Console.Out.WriteLine("p=(" + p.Prenom + "," + p.Nom + "," +
            p.Age + ")");
            try
            {
                p.Age = -4;
            }
            catch (Exception ex)
            {
                Console.Error.WriteLine(ex.Message);
            }//try-catch
        }
    }
}
```

Dans l'instruction *Console.Out.WriteLine("p=(" + p.Prenom + "," + p.Nom + "," + p.Age + ")");* on cherche à avoir les valeurs des propriétés P*renom*, N*om* et A*ge* de la personne **p**. C'est la méthode **get** de ces propriétés qui est alors appelée et qui rend la valeur de l'attribut qu'elles gèrent.

Dans l'instruction *p.Age=56;* on veut fixer la valeur de la propriété A*ge*. C'est alors la méthode **set** de cette propriété qui est alors appelée. Elle recevra 56 dans son paramètre **value**.

Une propriété **P** d'une classe **C** qui ne définirait que la méthode **get** est dite en lecture seule. Si c est un objet de classe C, l'opération *c.P=valeur* sera alors refusée par le compilateur.

6.1.11 Les méthodes et attributs de classe

Supposons qu'on veuille compter le nombre d'objets *Personne* créées dans une application. On peut soi-même gérer un compteur mais on risque d'oublier les objets temporaires qui sont créés ici ou là.

Il semblerait plus sûr d'inclure dans les constructeurs de la classe *Personne*, une instruction incrémentant un compteur. Le problème est de passer une référence de ce compteur afin que le constructeur puisse l'incrémenter : il faut leur passer un nouveau paramètre.

On peut aussi inclure le compteur dans la définition de la classe. Comme c'est un attribut de la classe elle-même et non celui d'une instance particulière de cette classe, on le déclare différemment avec le mot clé *static* :

```
private static long nbPersonnes;
```

Pour le référencer, on écrit P*ersonne.nbPersonnes* pour montrer que c'est un attribut de la classe P*ersonne* elle-même. Ici, nous avons créé un attribut privé auquel on n'aura pas accès directement en-dehors de la classe. On crée donc une propriété publique pour donner accès à

l'attribut de classe *nbPersonnes*. Pour rendre la valeur de *nbPersonnes*, la méthode *get* de cette propriété n'a pas besoin d'un objet P*ersonne* particulier : en effet *nbPersonnes* est l'attribut de toute une classe. Aussi a-t-on besoin d'une propriété déclarée elle aussi *static* :

```
        public static long NbPersonnes
        {
            get { return nbPersonnes; }
        }
```

qui de l'extérieur sera appelée avec la syntaxe P*ersonne.NbPersonnes*.

Voici un exemple.

La classe P*ersonne* devient la suivante :

```
using System;

namespace Chapitre6
{
    public class Personne
    {   // attributs de classe
        private static long nbPersonnes;
        public static long NbPersonnes
        {
            get { return nbPersonnes; }
        }

        // attributs d'instance
        private string prenom;
        private string nom;
        private int age;

        // constructeurs
        public Personne(String p, String n, int age)
        {
            Initialise(p, n, age);
            nbPersonnes++;
        }
        public Personne(Personne p)
        {
            Initialise(p);
            nbPersonnes++;
        }

        // méthode
        public void Initialise(string p, string n, int age)
        {
            this.prenom = p;
            this.nom = n;
            this.age = age;
        }
        public void Initialise(Personne p)
        {
            prenom = p.prenom;
            nom = p.nom;
            age = p.age;
        }
        …
    }
}
```

Avec le programme suivant :

```
using System;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne p1 = new Personne("Mohamed", "Salah", 30);
            Personne p2 = new Personne(p1);
            new Personne(p1);
            Console.WriteLine("Nombre de personnes créées : " +
            Personne.NbPersonnes);
        }
    }
}
```

on obtient les résultats suivants :

Nombre de personnes créées : 3

6.1.12 Un tableau de personnes

Un objet est une donnée comme une autre et à ce titre plusieurs objets peuvent être rassemblés dans un tableau :

```
using System;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            // un tableau de personnes
            Personne[] amis = new Personne[3];
            amis[0] = new Personne("Mohamed", "Salah", 30);
            amis[1] = new Personne("Slim", "Ben Slim", 52);
            amis[2] = new Personne("Neil", "Armstrong", 66);
            // affichage
            foreach (Personne ami in amis)
            {
                ami.Identifie();
            }
        }
    }
}
```

On obtient les résultats suivants :

[Mohamed, Salah, 30]
[Slim, Ben Slim, 52]
[Neil, Armstrong, 66]

6.1.13 Méthodes génériques

On rappelle qu'une méthode statique s'utilise en préfixant la méthode par le nom de la classe et non par celui d'une instance de la classe. La méthode Sort a différentes signatures (elle est surchargée). Nous utiliserons la signature suivante :
public static void Sort<T>(T[] tableau, IComparer<T> comparateur)

Sort une méthode générique où T désigne un type quelconque. La méthode reçoit deux paramètres :
- T[] tableau : le tableau d'éléments de type T à trier
- IComparer<T> comparateur : une référence d'objet implémentant l'interface IComparer<T>.

IComparer<T> est une interface générique définie comme suit :

```
    public interface IComparer<T>{
      int Compare(T t1, T t2);
}
```

L'interface IComparer<T> n'a qu'une unique méthode. La méthode Compare :
- reçoit en paramètres deux éléments t1 et t2 de type T
- rend 1 si t1>t2, 0 si t1==t2, -1 si t1<t2. C'est au développeur de donner une signification aux opérateurs <, ==, >.

Par exemple, si p1 et p2 sont deux objets Personne, on pourra dire que p1>p2 si le nom de p1 précède le nom de p2 dans l'ordre alphabétique. On aura alors un tri croissant selon le nom des personnes. Si on veut un tri selon l'âge, on dira que p1>p2 si l'âge de p1 est supérieur à l'âge de p2.
- pour avoir un tri dans l'ordre décroissant, il suffit d'inverser les résultats +1 et -1

Exemple :

```
using System;
using System.Collections.Generic  ;
namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {

            // un tableau de personnes
            Personne[] personnes1 = { new Personne("Ben Saleh", "Taoufik",
            25), new Personne("Troudi","Samir", 35), new
            Personne("Mustafa", "Samir", 32) };
            // affichage
            Affiche("Tableau a trier ", personnes1);
            // tri selon le nom
            Array.Sort(personnes1, new CompareNoms());
            // affichage
            Affiche("Tableau apres le tri selon les nom et prenom ",
            personnes1);
            // tri selon l'âge
            Array.Sort(personnes1, new CompareAges());
            // affichage
            Affiche("Tableau apres le tri selon l'age ", personnes1);
        }

        static void Affiche(string texte, Personne[] personnes) {
        Console.WriteLine(texte.PadRight(50, '-'));
        foreach (Personne p in personnes) {
            Console.WriteLine(p);
        }
        }
    }
```

```
    // classe de comparaison des noms et prénoms des personnes
    class CompareNoms : IComparer<Personne> {
    public int Compare(Personne p1, Personne p2) {
        // on compare les noms
        int i = p1.Nom.CompareTo(p2.Nom);
        if (i != 0)
        return i;
        // égalité des noms - on compare les prénoms
        return p1.Prenom.CompareTo(p2.Prenom);
    }}

    // classe de comparaison des ages des personnes
    class CompareAges : IComparer<Personne> {
    public int Compare(Personne p1, Personne p2) {
    // on compare les ages
        if (p1.Age > p2.Age)
            return 1;
        else if (p1.Age == p2.Age)
            return 0;
        else
            return -1;
    }
    }
}
```

`Array.Sort(personnes1, new CompareNoms()`: le tri du tableau de personnes selon les nom et prénom. Le 2ième paramètre de la méthode générique Sort est une instance d'une classe CompareNoms implémentant l'interface générique IComparer<Personne>.

Les résultats de l'exécution sont les suivants :

Tableau à trier----------
[Ben Saleh, Taoufik, 25]
[Troudi, Samir, 35]
[Mustafa, Samir, 32]

Tableau après le tri selon les nom et prénom------
[Mustafa, Samir, 32]
[Troudi, Samir, 35]
[Ben Saleh, Taoufik, 25]

Tableau après le tri selon l'âge----------
[Ben Saleh, Taoufik, 25]
[Mustafa, Samir, 32]
[Troudi, Samir, 35]

6.2 L'héritage par l'exemple

6.2.1 Généralités

Le but de l'héritage est de "personnaliser" une classe existante pour qu'elle satisfasse à nos besoins.

Supposons qu'on veuille créer une classe *Enseignant* : un enseignant est une personne particulière. Il a des attributs qu'une autre personne n'aura pas : la matière qu'il enseigne par exemple. Mais il a aussi les attributs de toute personne : prénom, nom et âge.
→Un enseignant fait donc pleinement partie de la classe *Personne* mais a des attributs supplémentaires.

Plutôt que d'écrire une classe *Enseignant* à partir de rien, on préfèrerait reprendre l'acquis de la classe *Personne* qu'on adapterait au caractère particulier des enseignants. C'est le concept.

Pour exprimer que la classe *Enseignant* hérite des propriétés de la classe *Personne*, on écrira :

public class Enseignant : Personne

Personne est appelée la classe **parent** (ou mère) et *Enseignant* la classe **dérivée** (ou fille).

Un objet *Enseignant* a toutes les qualités d'un objet *Personne* : il a les mêmes attributs et les mêmes méthodes. Ces attributs et méthodes de la classe parent ne sont pas répétées dans la définition de la classe fille : on se contente d'indiquer les attributs et méthodes rajoutés par la classe fille :
Nous supposons que la classe *Personne* est définie comme suit :

```
using System;
namespace Chapitre6
{
    public class Personne
    {    // attributs de classe
        private static long nbPersonnes;
        public static long NbPersonnes
        {
            get { return nbPersonnes; }
        }

        // attributs d'instance
        private string prenom;
        private string nom;
        private int age;

        // constructeurs
        public Personne(String prenom, String nom, int age)
        {
            Nom = nom;
            Prenom = prenom;
            Age = age;
            nbPersonnes++;
            Console.WriteLine("Constructeur Personne(string, string,
            int)");
        }
        public Personne(Personne p)
        {
            Nom = p.Nom;
            Prenom = p.Prenom;
            Age = p.Age;
            nbPersonnes++;
            Console.WriteLine("Constructeur Personne(Personne)");
        }

        // propriétés
        public string Prenom
```

```
        {
            get { return prenom; }
            set
            {
                // prénom valide ?
                if (value == null || value.Trim().Length == 0)
                {
                    throw new Exception("prénom (" + value + ") invalide");
                }
                else
                {
                    prenom = value;
                }
            }//if
        }//prenom

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

        public int Age
        {
            get { return age; }
            set
            {
                // age valide ?
                if (value >= 0)
                {
                    age = value;
                }
                else
                    throw new Exception("âge (" + value + ") invalide");
            }//if
        }//age
        // propriété
        public string Identite
        {
            get { return String.Format("[{0}, {1}, {2}]", prenom, nom,
age); }
        }
    }
}
```

La méthode *Identifie* a été remplacée par la propriété *Identite* en lecture seule et qui identifie la personne. Nous créons une classe *Enseignant* héritant de la classe *Personne* :

```
using System;

namespace Chapitre6
{
    class Enseignant : Personne
    {
```

```
        // attributs
        private int section;
        // constructeur
        public Enseignant(string prenom, string nom, int age, int section)
            : base(prenom, nom, age)
        {
            // on mémorise la section via la propriété Section
            Section = section;
            // suivi
            Console.WriteLine("Construction Enseignant(string, string, int,
            int)");
        }//constructeur

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

Tentons un premier programme de test [Program.cs] :

```
using System;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
  Console.WriteLine(new Enseignant("Mohamed", "Salah", 30, 27).Identite);
        }
    }
}
```

Ce programme ce contente de créer un objet *Enseignant* (new) et de l'identifier. La classe *Enseignant* n'a pas de méthode I*dentite* mais sa classe parent en a une qui de plus est publique : elle devient par héritage une méthode publique de la classe E*nseignant*.
Les résultats obtenus sont les suivants :

Constructeur Personne(string, string, int)
Construction Enseignant(string, string, int, int)
[Mohamed, Salah, 30]

6.2.2 Redéfinition d'une méthode ou d'une propriété

Dans l'exemple précédent, nous avons eu l'identité de la partie P*ersonne* de l'enseignant mais il manque certaines informations propres à la classe *Enseignant* (la section). On est donc amené à écrire une propriété permettant d'identifier l'enseignant :

```
1. using System;

2. namespace Chapitre6{
3. class Enseignant : Personne
4. {
5. // attributs
6. private int section;

7. // constructeur
```

```
8. public Enseignant(string prenom, string nom, int age, int section)
9. : base(prenom, nom, age)
10. {
11. // on mémorise la section via la propriété Section
12. Section = section;
13. // suivi
14. Console.WriteLine("Construction Enseignant(string, string, int,
    int)");
15. }//constructeur

16. // propriété Section
17. public int Section
18. {
19. get { return section; }
20. set { section = value; }
21. }// section

22. // propriété Identite
23. public new string Identite
24. {
25. get { return String.Format("Enseignant[{0},{1}]", base.Identite,
    Section); }
26. }
27. }
28. }
```

Ligne 24-26 : La propriété *Identite* de la classe *Enseignant* s'appuie sur la propriété *Identite* de sa classe mère (*base.Identite*) (ligne 25) pour afficher sa partie "P*ersonne*" puis complète avec le champ *section* qui est propre à la classe *Enseignant*. Notons la déclaration de la propriété *Identite* :

public new string Identite{

La propriété **Identite** est définie à la fois dans la classe *Enseignant* et sa classe mère *Personne*. Dans la classe fille *Enseignant*, la propriété *Identite* doit être précédée du mot clé **new** pour indiquer qu'on redéfinit une nouvelle propriété *Identite* pour la classe *Enseignant*.

public new string Identite{

La classe E*nseignant* dispose maintenant de deux propriétés I*dentite* :

- celle héritée de la classe parent P*ersonne*
- la sienne propre

Si E est un ojet E*nseignant*, *E.Identite* désigne la propriété I*dentite* de la classe E*nseignant*. On dit que la propriété I*dentite* de la classe fille **redéfinit** ou cache la propriété I*dentite* de la classe mère. De façon générale, si *O* est un objet et *M* une méthode, pour exécuter la méthode *O.M*, le système cherche une méthode *M* dans l'ordre suivant :

- dans la classe de l'objet O
- dans sa classe mère s'il en a une
- dans la classe mère de sa classe mère si elle existe
- etc…

L'héritage permet donc de redéfinir dans la classe fille des méthodes/propriétés de même nom dans la classe mère. C'est ce qui permet d'adapter la classe fille à ses propres besoins. Associée au polymorphisme que nous allons voir un peu plus loin, la redéfinition de méthodes/propriétés est le principal intérêt de l'héritage.

6.2.3 Le polymorphisme

Le fait qu'une variable *Oi* de classe *Ci* puisse en fait référencer non seulement un objet de la classe *Ci* mais en fait tout objet dérivé de la classe *Ci*, est appelé **polymorphisme** : la faculté pour une variable de référencer différents types d'objets.
Prenons un exemple et considérons la fonction suivante indépendante de toute classe (*static*):

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

On pourra aussi bien écrire

Personne p;
...
Affiche(p);

que

Enseignant e;
...
Affiche(e);

Dans ce dernier cas, le paramètre formel *p* de type *Personne* de la méthode statique A*ffiche* va recevoir une valeur de type *Enseignant*. Comme le type *Enseignant* dérive du type *Personne*, c'est légal.

6.2.4 Redéfinition et polymorphisme

Complétons notre méthode A*ffiche* :

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

La propriété *p.Identite* rend une chaîne de caractères identifiant l'objet *Personne p*. Que se passe-t-il dans l'exemple précédent si le paramètre passé à la méthode *Affiche* est un objet de type *Enseignant* :

Enseignant e = new Enseignant(...);
Affiche(e);

Regardons l'exemple suivant :

```
1. using System;

2. namespace Chapitre6 {
3. class Program {
4. static void Main(string[] args){
5. // un enseignant
6. Enseignant e = new Enseignant("Mounir", "Jendoubi", 56, 61);
7. Affiche(e);
8. // une personne
9. Personne p = new Personne("Mohamed", "Salah", 30);
10.      Affiche(p);
11.      }

12.      // affiche
13.      public static void Affiche(Personne p) {
14.      // affiche identité de p
15.      Console.WriteLine(p.Identite);
```

```
16.       }//affiche
17.       }
18.       }
```

Les résultats obtenus sont les suivants :

Constructeur Personne(string, string, int)
Construction Enseignant(string, string, int, int)
[Mounir, Jendoubi, 56]
Constructeur Personne(string, string, int)
[Mohamed, Salah, 30]

L'exécution montre que l'instruction *p.Identite* (ligne 15) a exécuté à chaque fois la propriété *Identite* d'une *Personne*, d'abord (ligne 6) la personne contenue dans l'*Enseignant* e, puis (ligne 9) la *Personne* p elle-même. Elle ne s'est pas adaptée à l'objet réellement passé en paramètre à *Affiche*.

On aurait préféré avoir l'identité complète de l'*Enseignant* e. Il aurait fallu pour cela que la notation *p.Identite* référence la propriété *Identite* de l'objet réellement pointé par *p* plutôt que la propriété *Identite* de partie "*Personne*" de l'objet réellement par *p*.

Il est possible d'obtenir ce résultat en déclarant *Identite* comme une propriété **virtuelle** (**virtual**) dans la classe de base *Personne* :

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

Le mot clé **virtual** fait de *Identite* une propriété virtuelle. Ce mot clé peut s'appliquer également aux méthodes. Les classes filles qui redéfinissent une propriété ou méthode virtuelle doivent alors utiliser le mot clé **override** au lieu de **new** pour qualifier leur propriété/méthode redéfinie. Ainsi dans la classe *Enseignant*, la propriété *Identite* est redéfinie comme suit :

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

Le programme précédent produit alors les résultats suivants :

Constructeur Personne(string, string, int)
Construction Enseignant(string, string, int, int)
Enseignant[[Mounir, Jendoubi, 56],61]
Constructeur Personne(string, string, int)
[Mohamed, Salah, 30]

La classe o*bject* (alias C# de *System.Object*) est la classe "mère" de toutes les classes C#. Ainsi lorsqu'on écrit :

public class Personne

on écrit implicitement :

public class Personne : System.Object

La classe *System.Object* définit une méthode virtuelle **ToString** :

La méthode *ToString* rend le nom de la classe à laquelle appartient l'objet comme le montre l'exemple suivant :

```
using System;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            // un enseignant
Console.WriteLine(new Enseignant("Mounir", "Jendoubi", 56, 61).ToString());
            // une personne
Console.WriteLine(new Personne("Mohamed", "Salah", 30).ToString());
        }
    }
}
```

Les résultats produits sont les suivants :

Constructeur Personne(string, string, int)
Construction Enseignant(string, string, int, int)
Chapitre6.Enseignant
Constructeur Personne(string, string, int)
Chapitre6.Personne

On remarquera que bien que nous n'ayons pas redéfini la méthode *ToString* dans les classes *Personne* et *Enseignant*, on peut cependant constater que la méthode *ToString* de la classe *Object* a été capable d'afficher le nom réel de la classe de l'objet.

Redéfinissons la méthode *ToString* dans les classes *Personne* et *Enseignant* :

```
// méthode ToString
public override string ToString()
{
  return Identite;
}
```

La définition est la même dans les deux classes. Considérons le programme de test suivant :

```
1.  using System;

2.  namespace Chapitre6
3.  {
4.  class Program
5.  {
6.        static void Main(string[] args)
7.        {
8.           Enseignant e = new Enseignant("Mounir", "Jendoubi", 56, 61);
9.           Affiche(e);
10.          // une personne
11.          Personne p = new Personne("Mohamed", "Salah", 30);
12.          Affiche(p);
13.       }
14.       // affiche
15.       public static void Affiche(Personne p)
16.       {
17.         // affiche identité de p
18.          Console.WriteLine(p);
19.       }//Affiche
    }
```

```
}
```

Attardons-nous sur la méthode *Affiche* qui admet pour paramètre une personne *p*. Ligne 18, la méthode *WriteLine* de la classe *Console* n'a aucune variante admettant un paramètre de type *Personne*.

Parmi les différentes variantes de *Writeline*, il en existe une qui admet comme paramètre un type *Object*.

Le compilateur va utiliser cette méthode, *WriteLine(Object o)*, parce que cette signature signifie que le paramètre *o* peut être de type *Object* ou dérivé. Puisque *Object* est la classe mère de toutes les classes, tout objet peut être passé en paramètre à *WriteLine* et donc un objet de type *Personne* ou *Enseignant*. La méthode *WriteLine(Object o)* écrit *o.ToString()* dans le flux d'écriture *Out*.

La méthode *ToString* étant virtuelle, si l'objet o (de type *Object* ou dérivé) a redéfini la méthode *ToString*, ce sera cette dernière qui sera utilisée. C'est ici le cas avec les classes *Personne* et *Enseignant*.

C'est ce que montrent les résultats d'exécution :

```
Constructeur Personne(string, string, int)
Construction Enseignant(string, string, int, int)
Enseignant[[Mounir, Jendoubi, 56],61]
Constructeur Personne(string, string, int)
[Mohamed, Salah, 30]
```

6.3 Redéfinir la signification d'un opérateur pour une classe

Considérons l'instruction

op1 + op2

où *op1* et *op2* sont deux opérandes. Il est possible de redéfinir la signification de l'opérateur +. Si l'opérande *op1* est un objet de classe *C1*, il faut définir une méthode statique dans la classe *C1* avec la signature suivante :

public static [type] operator +(C1 opérande1, C2 opérande2);

Lorsque le compilateur rencontre l'instruction

op1 + op2

il la traduit alors par *C1.operator+(op1,op2)*. Le type rendu par la méthode *operator* est important.

On peut redéfinir également les opérateurs unaires n'ayant qu'un seul opérande. Ainsi si *op1* est un objet de type *C1*, l'opération *op1++* peut être redéfinie par une méthode statique de la classe *C1* :

public static [type] operator ++(C1 opérande1);

Exemple

On crée une classe *ListeDePersonnes* dérivée de la classe *ArrayList*. Cette classe implémente une liste dynamique.. De cette classe, nous n'utilisons que les éléments suivants :

- la méthode L.*Add(Object o)* permettant d'ajouter à la liste *L* un objet *o*. Ici l'objet *o* sera un objet *Personne*.
- la propriété L.*Count* qui donne le nombre d'éléments de la liste L
- la notation L*[i]* qui donne l'élément i de la liste L

La classe *ListeDePersonnes* va hériter de tous les attributs, méthodes et propriétés de la classe *ArrayList*. Sa définition est la suivante :

```
using System;
using System.Collections;
using System.Text;

namespace Chapitre6
{
    class ListeDePersonnes : ArrayList
    {
      // redéfinition opérateur +, pour ajouter une personne à la liste
      public static ListeDePersonnes operator +(ListeDePersonnes l,
      Personne p)
      {
        // on ajoute la Personne p à la ListeDePersonnes l
        l.Add(p);
        // on rend la ListeDePersonnes l
        return l;
      }// operator +

      // ToString
      public override string ToString()
      {
        // rend (él1, él2, ..., éln)
        // parenthèse ouvrante
        StringBuilder listeToString = new StringBuilder("(");
        // on parcourt la liste de personnes (this)
        for (int i = 0; i < Count - 1; i++)
        {
           listeToString.Append(this[i]).Append(",");
        }//for
        // dernier élément
        if (Count != 0)
        {
           listeToString.Append(this[Count - 1]);
        }
        // parenthèse fermante
        listeToString.Append(")");
        // on doit rendre un string
        return listeToString.ToString();
       }//ToString
    }
}
```

Une classe de test pourrait être la suivante :

```
using System;
using System.Collections;
using System.Text;

namespace Chapitre6
{
    class Program
    {
        static void Main(string[] args)
        {
            // une liste de personnes
            ListeDePersonnes l = new ListeDePersonnes();
            // ajout de personnes
```

```
            l = l + new Personne("Mohamed", "Slim", 10) + new
Personne("Emna", "Foued", 12);
            // affichage
            Console.WriteLine("l=" + l);
            l = l + new Enseignant("Samia", "Mustapha", 27, 60);
            Console.WriteLine("l=" + l);
        }
    }
}
```

Les résultats :

l=([Mohamed, Slim, 10],[Emna, Foued, 12])

l=([Mohamed, Slim, 10],[Emna, Foued, 12],Enseignant[[Samia, Mustapha, 27],60])