Programmation Objet - C#

1 Introduction

1.1 Qu'est-ce que la programmation orientée objet ?

Contrairement à la programmation dite impérative, qui n'est qu'un simple traitement sur des données, la programmation orientée objet (ou POO) est une technique visant à faire interagir des objets entre eux, permettant une meilleure modularité et une plus grande souplesse de la programmation.

Ainsi, un objet va être constitué par l'association d'une quantité d'information organisée en champs (nom, prénom, age, notes pour un étudiant ; marque, modèle, cylindrée, vitesse pour une voiture) et d'un ensemble de méthodes (plus ou moins équivalentes à des fonctions) permettant d'interagir avec lui (calculer la moyenne d'un étudiant ou accélérer pour une voiture).

Cette technique de programmation a principalement deux objectifs :

- Faciliter l'écriture des applications par une structuration en terme d'objets.
- Favoriser la réutilisation de code, en composant des programmes à partir d'objets existants (par exemple, la bibliothèque standart de C++, de Java, ...).

Deux notions essentielles sont caractéristiques de la POO :

- La notion de classe : schématiquement, une classe représente le type d'un objet, tout comme *int* représente un réel. Elle peut être vu comme une sorte de super-structure.
- La notion de méthode : schématiquement, une méthode est une fonction appartenant à une classe et permettant d'agir sur et avec l'objet.

1.2 Concepts clefs de la programmation orientée objet

La POO a introduit des concepts permettant une bien meilleure souplesse de programmation ainsi qu'une plus grande sécurité.

Dans un soucis de réutilisation du code, une classe doit être sufisamment bien conçu pour qu'un programmeur n'ait pas à s'inquiéter de l'état interne d'un objet, mais se contente de savoir de quelle manière il interagit avec son environnement. Cette volonté de masquer la représentation de l'objet s'appelle l'encapsulation et amène aussi une sécurité du code dans la mesure ou l'état de l'objet ne peut être changé que de manière explicite. Celà permet aussi de modifier la représentation d'une classe sans forcément modifier les interactions et donc les programmes dépendants de ces classes.

Toujours dans un soucis d'efficacité et de modularité, la POO a introduit la notion d'héritage entre les classes. Très schématiquement, on peut définir grâce à l'héritage une classe dite « mère » à partir de laquelle on va définir plusieurs classes dites « filles » ayant en commun les caractéristiques de la classe mère. Cela permet une économie du code et favorise sa réutilisation.

La POO introduit enfin des concepts plus avancés tels que le polymorphisme et la généricité qui seront abordés ultérieurement.

1.3 Principales notions de base

1.3.1 Classe

Schématiquement, une classe est la « description » d'une collection objets homogènes. La notion de classe peut être utilisé de plusieurs façons :

- pour définir un ensemble de sous-programmes (classes utilitaires) ;
- pour définir un nouveau type d'objet, par exemple un objet de type voiture, de type personne ou de type élève.

Une classe doit nécessairement comporter les éléments suivants :

- les champs : ce sont les « variables » permettant, en général, de définir l'état de l'objet.
- un ou plusieurs constructeurs : ce sont des fonctions indiquant de quelle façon l'objet doit être déclaré et initialisé.
- les méthodes : ce sont les fonctions permettant d'agir sur les objets d'une classe. Selon leur définition, elle peuvent s'appliquer sur un objet particulier, ou sur l'ensemble des instances d'une classe.

1.3.2 Objet

Nous avons vu précédemment qu'un objet est une « instance » de classe. Par analogie, on peut aussi considérer une classe comme le type d'une variable et l'objet comme la valeur de cette même variable.

A sa création, un objet est contruit et initialisé grâce à un constructeur et il va rester en mémoire jusqu'à la fin de la fonction qui l'emploie, de la même façon qu'une variable. Au moment de sa destruction, un fonction spéciale, appellée un destructeur, est appelée, qui est ici masqué par le language C# (ainsi qu'en Java) contrairement à d'autre languages orientés objets comme le C++ qui nécessitent une définition formelle du destructeur au même titre que la définition d'un constructeur.

2 Notion de classe

Dans la suite de ce cours, nous allons considérer à titre illustratif une classe *Personne* dont on veut stocker les nom, prénom, âge et numéro de téléphone tout étant capable de modifier et d'afficher ces informations.

2.1 Généralités

De manière formelle, une classe déclare un ensemble d'injonctions communes à une famille d'objets (des attributs pour l'état de l'objet et des méthodes

représentant le comportement de l'objet). Un bon exemple de classe existante est la classe *string* ou encore la classe *list*.

En C#, une classe sera toujours définie de la manière suivante :

```
public class Personne
{
     //Déclaration des champs
...

     //Déclaration des contructeurs
...

     //Déclaration des méthodes
...
}
```

2.2 Champs

Comme dit plus haut, les champs représentent les « variables » traduisant l'état de l'objet (comme avec les champs des structures). Ces variables peuvent être de toutes natures : des entiers, des chaînes de caractères, des listes, d'autres objets, etc... Par convention, leur nom commencent toujours par une minuscule.

Dans le cas de notre classe personne, nous allons déclarer les champs suivants : nom, prénom, age et numéro de téléphone :

```
public class Personne
{
     private string nom;
     private string prenom;
     private int age;
     private int telephone;
}
```

Le mot clef *public* devant le mot clef *class* indique que la classe *Personne* est accessible depuis n'importe quelle partie du programme. Elle aura aussi son importance au moment ou nous aborderons l'héritage.

Le mot clef *private* devant les champs indique ici que ces variables ne sont pas accessibles en dehors de la classe personne. Un programmeur utilisant la classe personne ne pourra donc pas utiliser ces variables directement dans d'autres fonction sauf si *private* est remplacé par le mot clef *public*. Si aucun mot clef n'est précisé, *private* sera ajouté par défaut à la compilation.

2.3 Constructeurs

Un contructeur est une fonction appellée au moment de la création d'un objet à partir d'une classe. Il permet d'initialiser correctement cet objet, éventuellement à partir de paramètres.

Plusieurs contructeurs peuvent être définis.

```
public class Personne
{
```

```
    //Champs
    private string nom;
    private string prenom;
    private int age;
    private int telephone;

    //Constructeurs
    public personne(){} //Constructeur par défaut

    public personne(string nom, string prenom,
        int age, int telephone)
    {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
        this.telephone = telephone;
    }

    public personne(personne p)
    {
        this.nom = p.nom;
        this.prenom = p.prenom;
        this.age = p.age;
        this.telephone = p.telephone;
    }
}
```

Le mot clef *this* est une référence sur l'instance courante. Elle permet d'accéder à l'objet appelant depuis la méthode et aussi déviter les confusions lorsque deux variables ont le même nom. Par exemple, dans le deuxième constructeur, le fait d'écrire this.nom permet de ne pas confondre le champ *nom* de l'objet personne avec la variable *nom* passée en paramètre pour l'initialisation. Le troisième constructeur est un constructeur particulier dans le sens ou il crée un objet personne en copiant les informations contenues dans un autre objet personne. On appelle cela un constructeur par copie.

Le premier constructeur est aussi un peu particulier vu qu'il ne prend aucun paramètre en entrée. Il est cependant essentiel de le définir afin d'éviter les surprises au moment de l'exécution et il est aussi utile si on définit une classe-fille comme nous le verrons ultérieurement.

2.4 Propriétés

Nous avons vu qu'en rendant les champs privés, un programmeur utilisateur de la classe personne ne pourra pas lire, ni écrire de valeur dans ces variables. Le comportement de l'objet devient alors équivalent à celui d'une boite noire dont on ne peut pas connaitre l'état interne. Cependant, il se peut que le programmeur ait besoin de connaitre ces informations et ait aussi besoin de les mettre à jour. Pour éviter de violer le principe d'encapsulation, nous allons donc définir des moyens d'accéder à ces variables à travers des *propriétés*. Une propriété permet de définir des accès en consultation (*get*) et en modification (*set*). Un

accès en consultation ou en modification est possible uniquement si le *get* ou *set* correspondant est définie.

La syntaxe d'une propriété se définit de la manière suivante :

```
public int Age
{
     get
     {
          return this.age;
     }

     set
     {
          this.age = value;
     }
 }
```

Le mot clef *public* indique bien que la propriété peut être utilisée dans n'importe quelle partie du code et que le mot clef *int* indique bien que *get* renvoie une valeur de type *int* et que *set* attend une valeur de type *int*.

Un autre exemple avec le champ nom :

```
public string Nom
{
     get
     {
          return this.nom;
     }

     set
     {
          if (value == null || value.Trim().Length == 0)
          {
               Console.WriteLine("Nom Invalide");
               nom = null;

          }
          else
          {
               nom = value;
          }
     }
}
```

Ici, pour éviter qu'une erreur ne survienne au moment de l'execution, il vaut mieux vérifier que la paramètre passé à *set* soit effectivement initialisé et qu'il ait une longueur supérieure à 0.

2.5 Méthodes

Les méthodes sont des « sous-porgrammes » propres à agir sur les objets d'une classe. Elles sont assimilables à des fonctions sauf qu'elles sont généralement

appelés par des objets et agissent potentiellement sur eux.

2.5.1 Méthodes d'instance

Ces méthodes décrivent des opérations effectuées sur une instance particulière de la classe (c'est-à-dire sur un seul objet).

Un bon exemple est la méthode de la classe *string* (chaînes de caractères) *ToLower()* qui permet de passer toutes les lettres d'une chaîne en minuscules de la façon suivante :

```
string s = new string("Une souris Verte
             qui Courait dans l'herBe");
s = s.ToLower();
```

Cette opération va transformer le contenu de s en : *une souris verte qui courait dans le pré.*

2.5.2 Champs, méthodes et propriétés de classe

Les champs de classe sont des champs que les objets d'une même classe vont partager. De même, les méthodes et les propriétés de classe sont des méthodes qui vont affecter l'ensemble des objets d'une classe.

Reprenons notre classe Personne. Nous aimerions disposer d'un compteur permettant de connaitre en permanence combien d'instances de Personne il existe. Il suffit alors de déclarer un champ d'instance de la manière suivante :

```
    private static int nbPersonne;
```

Remarquez au passage le mot clef *static* qui indique que ce champ est un champ de classe.

Il faut alors redéfinir les constructeur de la manière suivante :

```
public class Personne
{
     //Constructeurs
     public personne(){nbPersonne++;} //Constructeur par défault

     public personne(string nom, string prenom,
          int age, int telephone)
     {
          this.nom = nom;
          this.prenom = prenom;
          this.age = age;
          this.telephone = telephone;
          nbPersonne++;
     }

     public personne(personne p)
     {
          this.nom = p.nom;
          this.prenom = p.prenom;
          this.age = p.age;
```

```
            this.telephone = p.telephone;
            nbPersonne++;
        }
}
```

De manière à accéder à la variable *nbPersonne*, on définit une propriété de classe permettant un accès en lecture seule. En effet, il est préférable de ne pas laisser de possibilité au programmeur de modifier cette variable autrement qu'en créant différentes personnes.

```
public static long NbPersonne
{
    get { return nbPersonne; }
}
```

Pour les méthodes, cela fonctionne de la même façon. Il suffit d'ajouter le mot clef *static* pour la déclarer comme étant *de classe*. Elle s'appliquera alors sur l'ensemble des instances d'une classe.

2.5.3 Surcharge de méthodes

Nous avons vu précedemment que nous pouvions donner plusieurs formes de constructeur qui ayant pour objectifs d'initialiser un objet, mais avec des comportements différents.

Le C# autorise le même genre de comportement pour les méthodes. On peut appeller plusieurs méthodes par le même nom (la logique veut que ces méthodes aient un comportement similaire) tout en leur donnant des paramètres différents. On appelle celà la *surcharge* de méthode. Cela s'applique aussi bien aux méthodes d'instance qu'aux méthodes de classes. Un exemple concret se trouve dans la classe string avec l'ensemble des méthodes *IndexOf(...)*, permettant de chercher l'index (ou la position dans la chaîne) d'un caractère ou d'une sous-chaîne de caractères.

```
//méthodes surchargées
public int IndexOf(char value)...
public int IndexOf(string value)...
public int IndexOf(char value, int startIndex)...
public int IndexOf(string value, int startIndex)...
```

La différence entre ces méthodes se fait par le nombre de paramètres, leur type et leur ordre.

2.6 Opérateurs

Les opérateurs définis pour une classe sont identifiés par des symboles tels que =, +, -, ==, ¡=, ... Cependant, d'un point de vue logique, ces opérateurs ne sont pas différents des fonctions. Ils doivent être définis de la même manière, cependant leur utilisation est plus intuitive.

Voici un exemple d'opérateur permettant de savoir si deux personnes sont identiques :

```
public static bool operator ==(Personne a, Personne b)
{
     if(a.nom == b.nom && a.prenom == b.prenom &&
          a.age == b.age && a.telephone == b.telephone)
          return 1;
     else
          return 0;
}
```

Remarquez l'utilisation du mot clef *static* car il s'agit d'une « méthode » de classe, étant donné que cet opérateur doit pouvoir s'appliquer indifféremment à n'importe quelles instances de la classe.

Dans le programme principale, il sera utilisé de la façon suivante :

```
//On suppose que deux instances de Personnes p1 et p2
//ont déjà été initialisée
if(p1 == p2)
     Console.WriteLine("p1 et p2 sont la même personne");
else
     Console.WriteLine("p1 et p2 sont deux personnes différentes");
```

2.7 Notions de mutabilité

Ce terme désigne le fait qu'une instance de classe puisse être modifiée ou pas au cours de l'exécution du programme (c'est-à-dire modifier ses champs). Par exemple dans le cas de notre classe Personne, le fait d'autoriser l'accès en écriture sur les champs rend la classe mutable. Il suffirait de supprimer cette possibilité pour rendre cette classe non mutable.

2.8 En exemple complet : la classe Personne

Voici l'implémentation complète de la classe personne définie tout au long de cette section :

```
public class Personne
{
     //Champ et propriété de classe
     private static int nbPersonne;

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

     //Champs d'instance
     private string nom;
     private string prenom;
     private int age;
     private int telephone;

     //Constructeurs
```

```
public personne(){nbPersonne++;} //Constructeur par défault

public personne(string nom, string prenom, int age, int telephone)
{
     this.nom = nom;
     this.prenom = prenom;
     this.age = age;
     this.telephone = telephone;
     nbPersonne++;
}

public personne(personne p)
{
     this.nom = p.nom;
     this.prenom = p.prenom;
     this.age = p.age;
     this.telephone = p.telephone;
     nbPersonne++;
}

//Propriété
public string Nom
{
     get
     {
          return this.nom;
     }

     set
     {
          if (value == null || value.Trim().Length == 0)
          {
               Console.WriteLine("Nom Invalide");
               nom = null;
          }
          else
          {
               nom = value;
          }
     }
}

public string Prenom
{
     get
     {
          return this.prenom;
     }

     set
```

```
     {
          if (value == null || value.Trim().Length == 0)
          {
               Console.WriteLine("Prénom Invalide");
               prenom = null;
          }
          else
          {
               prenom = value;
          }
     }
}

public int Age
{
     get
     {
          return this.age;
     }

     set
     {
          this.age = value;
     }
}

public int Telephone
{
     get
     {
          return this.telephone;
     }

     set
     {
          this.telephone = value;
     }
}

//Opérateur
public static bool operator ==(Personne a, Personne b)
{
     if(a.nom == b.nom && a.prenom == b.prenom && a.age == b.age && a.telephone == b.tel
          return 1;
     else
          return 0;
}

//Méthode
public string Identifie()
```

```
    {
        string s = "p=(" + this.prenom + "," + this.nom + "," + p.age + "," + p.telephone "
        return s;
    }
}
```

3 Notion d'objet

Par définition, un objet est une instance d'une classe. Tandis qu'une classe est une représentation abstraite, l'objet représente quelque chose de très concret vu qu'il est stocké en mémoire et qu'on peut agir sur lui par le biais des méthodes, propriétés et opérateurs. Un exemple pour illustrer ce propos : la plan d'une maison représente la classe et on peut contruire une multitude de maison sur ce modèle.

3.1 Création

La création d'un objet obéit à des règles logiques et syntaxiques particulière. En effet, si vous vous contentez de déclarer :

```
Personne p1;
```

rien ne va se passer. Il faut comprendre que vous vous êtes contenté de *déclarer* une variable de type *Personne*. Mais cette variable n'est rien de plus qu'une coquille vide. Les champs ne sont pas remplit et il est impossible d'utiliser les méthodes pour la simple et bonne raison que la Personne p1 n'existe pas encore.

Pour pouvoir utiliser cette variable à notre convenance, il va falloir *contruire* la Personne. Pour ce faire, nous utilisons la syntaxe suivante :

```
p1 = new Personne("Dupond", "Jacques", 24, 0601020304);
```

On pourrait contracter l'ensemble de la façon suivante :

```
Personne p1 = new Personne("Dupond", "Jacques", 24, 0601020304);
```

À partir de ce moment, l'objet est construit et on peut l'utiliser normallement.

Nous avions aussi défini une constructeur dit « par copie ». Voici la façon de l'utiliser :

```
Personne p2 = new Personne(p1);
```

On crée une Personne p2 en copiant toute les informations contenues dans p1.

3.2 Référence

3.2.1 Retour sur les variables

Dans un programme informatique, la manipulation de données se fait par l'intermédiaire de variables. Ces variables résident en mémoire centrale sur la *pile d'exécution*. Une variable de type primitif (char, int, bool, ...) contient directement sa valeur. Par contre, une variable d'un type plus complexe (tableau, string, list, objet) contient une *référence* qui va pointer sur des données présentes sur le *tas*.

3.2.2 Références particulières

Le mot clef *this* est utilisé exclusivement dans la définition d'une classe comme référence à l'objet courant. Ainsi, il ne peut être utilisé que dans des méthodes et des propriétés d'instance. L'utiliser dans des méthodes de classe serait un non-sens.

Le mot clef *null* représente quant à lui une absence de référence. C'est une manière d'expliciter qu'une variable est une coquille vide. Il peut être affecté à toute variable.

3.2.3 Retour sur new

Le mot clef *new* n'est utilisé que lors de la construction d'un nouvel objet. Il permet l'allocation dynamique de la mémoire nécéssaire à cette nouvelle instance dans le tas. Il renvoie ainsi une référence vers la nouvelle instance crée. Cette référence sera alors stockée dans la pile. Ce procédé rend chaque instance unique et toute modification d'une propriété sur une instance n'aura aucune conséquence sur une autre (à part si il s'agit de champs de classes).

3.2.4 Référence et affectation

Imaginons que nous déclarons deux variable Personne p1 et p2 :

```
Personne p1 = new Personne("Dupond", "Jean", 44, 0601020304);
Personne p2;
p2 = p1;
```

La question est de savoir ce qu'il va se passer au moment de p2 = p1. Première remarque, nous n'avons pas utilisé le mot clef *new* pour p2. Je n'ai donc pas construit de nouvel objet. Ainsi, les variables p2 et p1 « pointent » sur le même objet. Elles ont donc la même référence et on peut à ce moment utiliser indifféremment p1 ou p2 pour agir sur l'objet considéré. Retenez donc que affectation et construction ne sont pas la même chose.

3.2.5 Comparaison de référence

Dans le cas ou le programmeur n'aurait pas surchargé les opérateurs == et != d'une classe, ces deux opérateurs comparent deux références. Ainsi, dans l'exemple précédent, p1 == p2 retournera un résultat positif. Il faut cependant faire attention à ne pas confondre comparer les références avec comparer les valeurs des objets.

3.3 Utilisation

La syntaxe pour manipuler les champs et méthodes d'un objet suit le prototype suivant : [nom de l'objet].[nom de la méthode ou du champ]. Par exemple, dans le cas de l'objet p1 de type Personne : p1.nom.

3.3.1 Utilisation des champs

Nous avions déclaré tous les champs comme *private*. Ce qui fait qu'à la compilation, la moindre occurence du type p1.nom, p2.telephone, ... se traduira par une erreur de compilation pour violation du principe d'encapsulation. Une solution serait de déclarer ces champs comme *public*. Cepedant, nous avions utiliser le terme *private* pour des raisons de sécurité et pour rendre le code plus fonctionnel. Pour accéder aux champs, nous allons donc devoir utiliser les propriétés qui ont pour mérite d'expliciter la lecture et l'écriture des champs.

```
Personne p = new Personne("Jean", "Michelin", 34, 0601020304);
Console.Out.WriteLine("p=(" + p.Prenom + "," +
     p.Nom + "," + p.Age + ")");
p.Age = 56;
Console.Out.WriteLine("p=(" + p.Prenom + "," +
     p.Nom + "," + p.Age + ")");
```

La définition de méthodes *get* au sein de ces propriétés permet de lire les champs et la définition de méthodes *set* permet tout simplement de modifier la valeur de ces champs. Dans l'exemple précédent, l'absence de *set* aurait entrainé une erreur de compilation pour p.Age = 56 et l'absence de methode *get* aurait empêché l'affichage sur la console.

3.3.2 Utilisation des méthodes et des opérateurs

Les opérateurs définis ou surchargés dans des classes s'utilisent exactement de la même façon que des opérateurs normaux. Il suffit de faire bien attention à leur définition et à leur fonctionnement et de les rendre le plus intuitifs possible. Par exemple, définir un opérateur ¿ dans la cas de la classe *Personne* n'aurait aucun sens, alors qu'il en aurait un pour une classe *Fraction*.

Concernant les méthodes, elles s'utilisent de la même façon qu'une fonction normale avec passage de paramètres et un type de sortie. Par exemple, la méthode identifie sera utiliser de la façon suivante :

```
Personne p = new Personne("Jean", "Michelin", 34, 0601020304);
Console.WriteLine(p.Identifie());
```

La fonction WriteLine demande un paramètre string, fournit par la sortie de la méthode Identifie.

Néanmoins, il faut garder en mémoire que les méthodes sont « appelées » par un objet et font donc partie d'un écosystème particulier.

3.4 Exemple

Voici un exemple d'utilisation de la classe Personne :

```
Personne p1 = new Personne("Jean", "Michelin", 34, 0601020304);
Personne p2 = new Personne("Jean", "Micheline", 34, 0601020305);

Personne p3 = new Personne(p1);
Personne p4 = p2;

Console.WriteLine(p1.Identifie());
Console.WriteLine(p3.Identifie());

p3.Age = 44;

Console.WriteLine(p1.Identifie());
Console.WriteLine(p3.Identifie());

Console.WriteLine(p2.Identifie());
Console.WriteLine(p4.Identifie());

p2.Age = 44;

Console.WriteLine(p2.Identifie());
Console.WriteLine(p4.Identifie());
```

Exercice : essayer de prévoir ce que le programme va afficher dans la console.

4 Héritage

4.1 Concepts

L'héritage est une technique permettant de définir une classe à partir d'une classe existente. Par exemple, à partir de la classe Personne apportant déjà les champs *nom, prenom, age, telephone* ainsi que quelques propriétés et méthodes, nous allons pouvoir définir d'autres classes dite *classes-filles*. Par exemple, si l'on veut modéliser le fonctionnement d'une université, nous pourrons créer les classes étudiants, professeurs, personnels techniques,... ou pour modéliser un avion de ligne, nous pourrons creer les classes pilotes, hôtesses et passagers. Ce mécanisme est une spécialisation, ou une généralisation selon les cas, à partir d'une classe mère regroupant leurs propriétés communes.

L'héritage est donc un mécanisme de réutilisation du code ou seul les différences entre les classes sont explicitées. On parle alors de *classe mère* et de *classe fille*.

Une chose à retenir en C#, l'héritage est dit : *héritage simple* car une classe fille ne peut peut avoir qu'une seule classe mère (comme en Java) à la différence d'autres languages comme le C++.

Une dernière chose permise par l'héritage est la redéfinition de méthodes n'ayant pas le fonctionnement souhaité dans la classse fille. La méthode est alors redéfinie pour adapter son comportement.

Voici la syntaxe pour déclarer qu'une classe hérite de la classe mère *Personne* :

```
public class Passager : Personne
{
```

```
...
}
```

C'est le symbole *:* qui permet d'expliciter que la classe Passager et l'héritière de la classe Personne.

4.2 Champs

Une classe fille conserve l'ensemble des champs de la classe mère mais peut néanmoins ajouter d'autres champs. Par exemple, dans le cas d'un passager, on pourrait rajouter son numéro de siège et le prix de son billet.

```
public class Passager : Personne
{
     //Champs
     private int siege;
     private double prix_billet;
}
```

Dans la cas d'un pilote, on pourra rajouter son nombre d'heures de vol ainsi que son salaire.

```
public class Pilote : Personne
{
     //Champs
     private int heures_vol;
     private double salaire;
}
```

Cependant, il faut faire attention à l'application du principe d'encapsulation. Même si les classes *Passager* et *Pilote* sont très proches de la classe *Personne*, le fait que les champs définis dans *Personne* le soit avec le mot clef *private* les oblige à utiliser les propriétés pour y accéder. Ce mode de fonctionnement n'étant pas très pratique, il existe un autre mot clef permettant l'accès des champs d'une classe à ses classes filles : *protected*. Cela reviendrait à définir la classe *Personne* de la manière suivante :

```
public class Personne
{
     //Champs
     protected string nom;
     protected string prenom;
     protected int age;
     protected int telephone;
}
```

De cette manière, les classes *Passager* et *Pilote* peuvent utiliser ces champs directement, facilitant la programmation.

Retenez surtout que *protected* permet l'accès au champs uniquement aux classes-filles de *Personne*, tandis que *public* autorise cet accès à l'ensemble du programme et que *private* restraint cet accès aux méthodes des objets de types *Personne*.

4.3 Constructeur

Contrairement aux méthodes, champs et propriétés, une classe fille n'hérite pas des constructeur de sa classe mère. Les constructeurs doivent donc être redéfinis. On peut cependant faire appel aux constructeurs de la classe mère dans un soucis de réutilisation du code.

```
public Passager(string nom, string prenom, int age, int telephone,
     int siege, double prix_billet) :
     base(nom, prenom, age, telephone)
{
     this.siege = siege;
     this.prix_billet = prix_billet;
}
```

Le mot clef *base* appelle le constructeur correspondant de la classe *Personne*. Si l'appel par le mot clef *base* est absent, alors le compilateur va se tourner vers un constructeur sans paramètre de *Personne*, (qui doit être préalablement définis) et renverra une erreur si ce contructeur n'existe pas.

Retenez bien qu'un objet Passager est aussi une Personne et doit donc être instancié comme une Personne avant de pouvoir être déclarer comme un Passager.

4.4 Méthodes

Certaines méthodes d'une classe mère peuvent ne plus être adaptée à sa/ses classe(s) fille(s). Il faut alors redéfinir la méthode en question.

Par exemple, la méthode *Identifie()* de la classe *Personne* ne permet pas de rendre compte de l'ensemble des informations présentes dans la classe *Passager*. Il nous faut donc la compléter de la manière suivante :

```
public string Identifie()
{
     string s = base.Identifie() + "," + this.siege + "," +
          this.prix_billet +  ")";
     return s;
}
```

Encore une fois, notez l'utilisation du mot clef *base* de manière à réutiliser la méthode de la classe mère. on pourrait très bien se passer de l'utilisation de ce mot clef, l'essentiel étant que la méthode Identifie soit adaptée à la classe On aurait tout aussi bien écrire cette méthode de la façon suivante :

```
public string Identifie()
{
     string s = "p=(" + this.prenom + "," + this.nom + ","
          + this.age + "," + this.telephone + "," +
          this.siege + "," + this.prix_billet + ")";
     return s;
}
```

Il faudra aussi penser à redéfinir les opérateurs si nécessaire.

4.5 Remarques complémentaires

La relation induite par l'héritage simple permet de constituer un arbre d'héritage. Cet outil doit être considéré comme une aide à la conception d'un programme permettant de hiérarchiser ses classes et de voir si certaines peuvent faire l'objet d'une classe mère commune permettant un gain de temps certain au niveau de la programmation.

Une dernière précision concernant le vocabulaire : une classe fille est aussi appellée une *sous-classe* et une classe mère, une *superclasse*.

4.6 Exemple

Pour la classe mère, reportez au code de la classe *Personne* définis à la deuxième section. Seul les champs d'instance sont déclarés comme *protected* plutôt que *private*.

```
public class Passager : Personne
{
     //Champs
     private int siege;
     private double prix_billet;

     //Constructeur
     public Passager(string nom, string prenom, int age,
          int telephone, int siege, double prix_billet) :
          base(nom, prenom, age, telephone)
     {
          this.siege = siege;
          this.prix_billet = prix_billet;
     }

     //Propriétés
     public int Siege
     {
          get
          {
              return this.siege;
          }

          set
          {
               this.siege = value;
          }
     }

     public double Prix_Billet
     {
          get
          {
                return this.prix_billet;
          }
```

```
            set
            {
                this.prix_billet = value;
            }
        }

        //Opérateur
        public static bool operator ==(Passager a, Passager b)
        {
            if(a.nom == b.nom && a.prenom == b.prenom &&
                a.age == b.age && a.telephone == b.telephone &&
                a.siege == b.siege && a.prix_billet == b.prix_billet)
            {
                return 1;
            }
            else
                return 0;
        }

        //Méthode
        public string Identifie()
        {
            string s = base.Identifie() + "," + this.siege + "," +
                this.prix_billet +  ")";
            return s;
        }
}
```

Littérature

- http://www.csharp-station.com
- http://csharp.net-tutorials.com
- Serge Tahé. Apprentissage du langage C# 2008 et du Framework .NET 3.5 (2008). (http://tahe.developpez.com/dotnet/csharp/)