1 Module B.1 : Fondamentaux de l'objet

Comme indiqué dans l'introduction de la première partie de cette formation, le C# est un langage orienté objet typé statiquement. La gestion des données en mémoire reposait dans la partie A sur la notion de type, mais l'on a fait abstraction de la partie objet du C#. En réalité, les précédents modules ont expérimenté la manipulation d'objets. Par exemple, le type **string** ou **DateTime** sont des classes fort utile pour gérer des chaînes de caractères et des dates. Les modules suivants présentent les principes fondamentaux de la programmation-objet pour ensuite illustrer celle-ci en C#.

1.1 Un paradigme de programmation

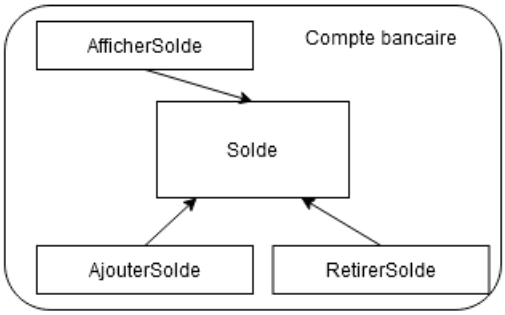
1.1.1 Bref Historique

Le paradigme de la programmation orientée objet a été élaboré au début des années 1960 par les Norvégiens Ole-Johan Dahl, Kristen Nygaard (langage Simula en 1962, Simula 67 étant le premier langage à classes) et poursuivi par l'américain Alan Kaye dans les années 1970 avec le premier langage orienté objet « SmallTalk ». Le développement fort rapide de l'utilisation de ce paradigme fut dû au développement au cours des années 1980 du C++ par le danois Bjarne Stroustrup. Ce dernier était construit sur le langage C, originalement nommé « C with Classes » et entièrement compatible avec ce dernier (il l'est toujours). Aujourd'hui, le paradigme orienté objet est extrêmement répandu et de nombreux langages l'incorporent dont le C#.

1.1.2 Conception informatique différente

Le paradigme de la programmation orientée objet impose une approche différente sur la conception d'une solution informatique. En programmation procédurale, comme en C ou en COBOL, il s'agit d'effectuer une série de traitements sur une série de données. Les procédures agissent sur les données et sont maîtresses. Dans une approche objet, c'est précisément l'inverse. Il s'agit en premier lieu de se concentrer sur les données et d'établir les traitements qu'il est autorisé de réaliser sur celles-ci.

Un objet est la conjugaison entre la donnée représenter sous la forme d'*champs* et de manipulations autorisées sur cette donnée que l'on appelle *méthodes*. Il peut représenter des objets de notre réalité (nous verrons dans les chapitres suivants les limites d'une telle pensée) comme un arbre, une voiture ou un concept abstrait comme un intervalle de temps. Le schéma ci-dessous présente schématiquement cette idée pour un objet appelé « Compte bancaire ». Dans cet exemple la donnée que nous manipulons est le solde. En réfléchissant sur les opérations pouvant manipuler ce solde bancaire, on détermine les méthodes de l'objet. Dans ce cas précis, on peut faire un virement pour incrémenter le solde, faire un retrait qui décrémente le solde et afficher le solde actuel.



Une classe est le plan d'un objet décrivant sa donnée et les opérations permettant de la manipuler. Elle définit un type d'objet. La classe permet de déclarer des instances d'un type d'objet. Chaque instance a son existence propre. Dans l'exemple précédent, la manipulation d'un objet de type « Compte bancaire » ne concerne que la donnée contenue dans l'instance concernée. Les autres instances ne sont pas modifiées par cette manipulation.

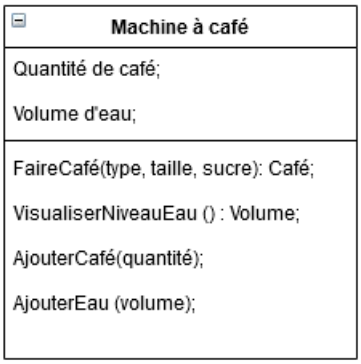
Exemple : Instances de type « Chaise »

1.1.3 Encapsulation

Pour un type d'objet donné, on s'est concentré sur les opérations permettant de manipuler la donnée de l'objet. A présent, il est important de faire la différence entre le développeur de la classe et son utilisateur. Une fois que l'on a défini les canaux permettant d'interagir avec les données, il est important que l'utilisateur de la classe ne manipule les données qu'en passant par les canaux à sa disposition. Le processus permettant d'interdire l'accès direct aux données est l'encapsulation. L'objet se comporte comme une boîte noire qui cache ses données (ses *champs*) et son fonctionnement interne pour l'utilisateur. L'utilisateur n'a accès qu'aux canaux de manipulation que sont les *méthodes*.

On prend l'exemple commun de la machine café. On n'a pas besoin de comprendre le processus interne d'une machine à café pour utiliser ses services. Elles disposent d'outils de manipulation comme des boutons permettant de choisir le type, la taille du café, la quantité de sucre. De plus, elle interagit avec l'utilisateur en indiquant le problème quand elle ne peut pas produire le café. On peut rajouter du café par une petite trappe, contrôler le niveau d'eau et en rajouter le cas échéant. Cet exemple est illustré ci-dessous avec un schéma simplifié de cette classe et un exemple d'instance de cette classe.

Exemple : Schéma et instance de type « Machine à café »



L'encapsulation permet d'utiliser des objets sans se soucier de leur fonctionnement interne. Cela permet un haut niveau d'abstraction des programmes informatiques. Dans l'idéal les *champs* ne doivent pas être accessibles directement et les *méthodes* sont les seuls composants que l'utilisateur peut utiliser.

1.2 L'objet en C#

On présente dans cette section la concrétisation des principes du paradigme de la programmation orientée objet présentés précédemment.

1.2.1 Définition et Utilisation

La déclaration d'une classe se réalise à l'aide du mot clé **class** suivi du nom de la classe puis de son corps. Le corps de la classe contient l'ensemble de ses champs et de ses méthodes. Un champ est une variable associée à une instance de la classe et une méthode est une fonction associée à une classe et ayant accès à ses paramètres, mais aussi aux champs de son instance.

```
modificateurs class Nom
{
  modificateurs type Nom champ 1;
  ...
  modificateurs type Nom champ N;

  modificateurs type retour Nom méthode 1 (paramètres)
  {
    // corps méthode 1
  }
  ...
}
```

Les noms doivent être composés de termes alphanumériques, pas de caractères spéciaux. Les mots commencent par une majuscule sauf indication contraire. Les règles sur le choix des noms sont les suivantes :

— Les champs privés commencent par _. Le premier mot ne prend pas de majuscule.

— Les champs non privés, les méthodes et les noms de classe commencent par un mot avec majuscule.

— Les paramètres de méthodes commencent par un mot sans majuscule.

Dans le diagramme ci-dessus, devant la définition de la classe, des champs et des méthodes, l'on peut observer le terme *modificateurs* désignant l'ensemble des mots clés permettant de modifier les caractéristiques du sujet de la modification. Les modificateurs sont tous optionnels.

Modificateurs d'accès

Dans la liste des modificateurs disponibles, il existe les modificateurs d'accès. Ces derniers permettent d'illustrer le concept d'encapsulation pour une classe. Chaque modificateur désigne la portée d'accès du sujet.

Le tableau suivant décrit les modificateurs d'accès disponibles.

Modificateur d'accès	Explications
public	Le sujet est visible et accessible par tous.
internal	Le sujet est visible et accessible par tous les éléments de son espace de nom.
protected	Le sujet est visible et accessible à l'intérieur de sa classe ou de ses classes dérivées.
private	Le sujet est visible et accessible à l'intérieur de sa classe.

Le modificateur d'accès par défaut est **internal**. Son intérêt repose sur le fait qu'il dispose des mêmes avantages que **public** dans le cadre de son espace de nom. Cela est intéressant pour des classes techniques qui servent le travail des classes fonctionnelles, disponibles pour l'utilisateur en bout de chaîne. Dans l'absolu, les champs et les méthodes techniques devraient être **private** et les méthodes fonctionnelles **public**. Le mot clé **protected** est abordé dans le module sur l'héritage.

Exemple

Pour reprendre l'exemple simple du compte bancaire, le code suivant permet de définir une classe le représentant.

```
1  // classe accessible à tous
2  public class CompteBancaire
3  {
4    // Champ.
5    private decimal _solde;
6
7    // Méthodes fonctionnelles.
8    public decimal AfficherSolde()
9    {
10     return _solde;
11   }
12
13   public void AjouterSolde (decimal somme)
14   {
15     if (somme < 0)
16       throw new ArgumentOutOfRangeException ("Un virement doit être
             positif.");
17
18     _solde += somme;
19   }
20
21   public void RetirerSolde (decimal somme)
22   {
23     if (somme > _solde)
24       throw new ArgumentOutOfRangeException ("Le retrait n'est pas
             possible.");
25
26     _solde -= somme;
27   }
28 }
```

Surcharge méthodes

Plusieurs méthodes peuvent coexister au même endroit si elles disposent d'une signature différente et que leurs types de retours sont différents. La signature prend en compte la liste des types des paramètres ainsi que l'ordre de ces paramètres.

Exemple : Surcharge de méthodes

```
1  public class CompteBancaire
2  {
3    ...
4    // Méthode RetirerSolde surchargée.
5    public void RetirerSolde (decimal somme) // (decimal)
6    {
7      if (somme > _solde)
8        throw new ArgumentOutOfRangeException ("Le retrait n'est pas
             possible.");
9
10     _solde -= somme;
11   }
12
13   /* Erreur compilation : seule la valeur de retour change.
14   public decimal RetirerSolde (decimal somme) // (decimal)
15   {
16     // Corps
17   }*/
18
19   // (decimal, decimal).
20   public decimal RetirerSolde (decimal somme, decimal minimum)
21   {
22     if (somme > _solde && minimum > _solde)
23       throw new ArgumentOutOfRangeException ("Le retrait n'est pas
             possible.");
24
25     decimal montant;
26
27     if (somme > _solde)
28     {
29       montant = _solde;
30       _solde = 0;
31     }
32     else
33     {
34     _solde -= somme;
35     montant = somme;
36     }
37
38     return montant;
39   }
40   ...
41 }
```

Il faut faire attention aux ambiguïtés de certaines surcharges de méthodes, notamment dues aux conversions implicites des types entre eux. Le compilateur produit une erreur lorsqu'il n'est pas capable de déterminer la méthode à choisir.

Exemple : Problème surcharge de méthodes

```
1  public class Point
2  {
3    private int x, y;
4
5    public void deplace (int dx, short dy) // (int, short).
6    {
7      x += dx; y += dy;
8    }
9    public void deplace (short dx, int dy) // (short, int).
10   {
11     x += dx; y += dy;
12   }
13 }
14 ...
15 Point point = ...;
16 int a, short b;
17 point.deplace (a, b); // (int, short).
18 point.deplace (b, a); // (short, int).
19 point.deplace (b, b); // Erreur car short -> int possible.
```

Déclaration et initialisation d'une instance

Une fois la définition de la classe réalisée, une instance peut être déclarée en utilisant le type de la classe comme n'importe quel type déjà vu précédemment. Les classes étant des types par référence, il faut utiliser le mot clé **new** pour initialiser l'instance.

Exemple : Déclaration et initialisation

```
1  CompteBancaire compte = new CompteBancaire ();
```

1.2.2 Constructeurs

L'initialisation d'une instance de classe est prise en charge par ce que l'on appelle un constructeur. Ce dernier est défini comme une méthode possédant le même nom que sa classe. Les paramètres de ce dernier permettent d'initialiser les champs de l'instance. Il ne possède pas de type de retour, l'utilisation de **void** est proscrite. Si aucun constructeur n'est défini, le constructeur sans paramètres est appelé. Ce dernier n'a pas besoin d'être défini s'il n'existe pas d'autres constructeurs. Si un constructeur est défini, l'initialisation d'une instance doit nécessairement appeler l'un des constructeurs publics définis.

Les constructeurs pour les structures ont déjà été abordés. La seule différence repose sur le fait que les constructeurs ne sont pas obligés d'initialiser tous les champs de l'instance. De plus les classes permettent de donner une valeur par défaut aux champs au moment de leur déclaration.

L'initialisation des champs se fait en trois étapes :

— Lors de la déclaration, valeur par défaut du type du champ est affecté à celui-ci.

— Si une valeur est définie par défaut, cette dernière est affectée au champ.

— Le constructeur peut affecter une valeur à ce champ.

Exemple : Constructeur sur compte bancaire

```
1  // Classe accessible à tous.
2  public class CompteBancaire
3  {
4    // Champ, valeur par défaut.
5    private decimal _solde = 0;
6
7    // Constructeur avec un paramètre.
8    CompteBancaire (decimal solde)
9    {
10     _solde = solde;
11   }
12   ...
13 }
```

Surcharge constructeurs

Les constructeurs comme les autres méthodes peuvent être surchargés. L'appel d'un constructeur par un autre se réalise à l'aide du mot clé **this**.

```
nom Classe (paramètres constructeur A) :
{
  // Corps constructeur A
}
...
nom Classe (paramètres constructeur B) : this (paramètres constructeur A)
{
  // Corps constructeur B
}
```

Exemple : Surcharge du constructeur sur compte bancaire

```
1  // Classe accessible à tous.
2  public class CompteBancaire
3  {
4    // Champ, valeur par défaut.
5    private decimal _solde = 0;
6    // Champ, sans valeur par défaut.
7    private string _possesseur;
8
9    // Constructeurs.
10   CompteBancaire (string nom)
11   {
12     _possesseur = nom;
13   }
```

```
14
15    CompteBancaire (string nom, decimal solde) : this (nom)
16    {
17     _solde = solde;
18    }
19    ...
20  }
```

1.2.3 Propriétés

Il est fort recommandé de laisser les champs en privés afin de cacher le fonctionnement interne des objets. Il est souvent utile de fournir la possibilité d'accéder à certains champs importants. Dans les autres langages-objets, il est coutume de fournir à ces champs des méthodes **get** et **set**, appelés *accesseurs* permettant de lire et de modifier respectivement ces champs. En C#, il existe un mécanisme permettant de réaliser cela, à savoir les propriétés qui ont la particularité pour l'utilisateur de se comporter comme des champs accessibles qui contiennent de la logique, comme les méthodes.

Les propriétés se déclarent comme des champs avec l'ajout d'une paire des blocs **get** et **set**. L'accesseur **get** est exécuté lorsque la propriété est lue, ce dernier doit nécessairement retourner une valeur du type de la propriété. L'accesseur **set** est exécuté lorsque la propriété est assignée. Un paramètre implicite nommé **value** du type de la propriété désigne la valeur à assigner.

Exemple : Propriétés

```
1   // Classe accessible à tous.
2   public class PointPositif
3   {
4     // Champ privé
5     private double _x;
6     private double _y;
7
8     // Propriétés
9     public double X
10    {
11      get { return _x; } // accesseur en lecture.
12      set                // accesseur en écriture.
13      {
14        if (x > 0)       // Logique interne.
15          _x = value;
16        else
17          _x = 0;
18      }
19    }
20    ...
21  }
```

Propriétés en lecture seule

On peut tout à fait utiliser en propriété en lecture seule en ne déclarant qu'un accesseur **get** ou en écriture seule avec un accesseur **set**. Le deuxième cas est fort rare, le premier est plus courant et permet d'éviter des méthodes superflues d'accès à des propriétés importantes sans pour autant laisser un accès libre aux champs encapsulés. Une propriété n'est pas nécessairement associée à un champ particulier (par exemple la distance à l'origine d'un point dans un plan euclidien).

Exemple : Propriété en lecture seule

```
1  // Classe accessible à tous.
2  public class CompteBancaire
3  {
4    // Champ privé.
5    private decimal _solde
6    ...
7    // Propriété remplaçant l'affichage du solde.
8    public decimal Solde
9    {
10     get { return _solde; } // accesseur en lecture.
11   }
12   ...
13 }
```

Propriétés auto-implémentées

La forme la plus classique de propriété consiste en une simple lecture et une simple écriture d'une valeur. Dans ce cas, il existe l'implémentation automatique d'une propriété. Si la propriété est en lecture seule, on peut rajouter un modificateur d'accès **private**
La forme est la suivante : `modificateur nom Propriété { get; set; }`

Exemple : Propriété auto-implémentée

```
1  // Classe accessible à tous.
2  public class CompteBancaire
3  {
4    // Champ privé.
5    private decimal _solde
6    ...
7    // Propriété remplaçant l'affichage du solde .
8    public decimal Solde { get; private set; }
9    ...
10 }
```