Programmation Orientée Objet en C#

1 Introduction

1.1 Présentation

Tout bon développeur le sait, le code d'un programme doit être propre, commenté, facile à maintenir et à améliorer.

Vous êtes adepte de la programmation procédurale et souhaitez apprendre la programmation objet ? Alors ce tutoriel est fait pour vous.

A travers celui-ci, vous découvrirez toutes les notions, aussi bien basiques qu'avancées, de la POO (Programmation Orientée Objet). Ces notions sont pour la plupart communes à tous les langages Objet (C#, Java, PHP, etc).

Voici un avant-goût de ce qui vous attend :

- Classe et objet
- Attribut et visibilité
- Constructeur et méthodes
- Héritage
- Interface et classes abstraites
- Classe et méthodes génériques
- Classe scellée

En vrac, on peut également parler d'initialiseur, d'indexeurs, de surcharge de méthodes et d'opérateurs, de classes partielles, d'expressions lambda, de types anonymes, etc.

1.2 Le langage C#

Pour illustrer nos propos, nous utiliserons le langage C# (Prononcez C-sharp). Sachez néanmoins que ce tutoriel peut être suivi indépendamment du langage de programmation utilisé puisque que, comme dis plus haut, les notions de programmation objet sont les mêmes pour tous les langages.

C# est un langage apparu en 2001 et spécialement crée par Microsoft pour être utilisé avec le Framework .NET. Microsoft a également sorti durant cette même année Visual Studio, son IDE (Integrated Development Environement) dédié au .NET.

Une nouvelle version, nommé Visual Studio 2010, sera disponible en avril prochain et introduira également le framework .NET 4.0

1.3 Prérequis

Première chose dont nous aurons besoin pour réaliser ce tutoriel, le framework .NET, évidemment. La version 3.5 SP1 est actuellement la dernière en date. Vous pouvez la télécharger directement depuis le centre de téléchargement Microsoft.

Les utilisateurs de Windows 7 n'auront pas besoin de le télécharger ni de l'installer puisque ce dernier est intégré de base dans le système.

Deuxième élément, il nous faut bien évidemment un outil de développement. Microsoft met à disposition de tous, gratuitement, Visual Studio C# Express 2008, qui est une sous-version de Visual Studio spécifique au C#.

Vous pouvez également vous procurer la version RC de Visual Studio 2010 gratuitement. Attention, il ne s'agit pas la version finale, qui elle, sera payante.

Si vous n'avez jamais fais de programmation orientée objet, il se peut fortement que ce tutoriel vous paraisse complexe. Il y a beaucoup de notions à retenir et je vous conseille vivement de pratiquer en même temps que vous suivez ce tutoriel.

Par ailleurs, vous devez déjà avoir fais de la programmation avant d'attaquer ce tutoriel, nous ne reviendrons pas sur les notions de fonctions, paramètres, variables, etc.

Pour vous faciliter la tâche, vous pouvez dès maintenant récupérer le projet Visual Studio qui contient les différents fichiers que j'utiliserai tout au long de ce tutoriel.

Si l'image n'apparait pas, cliquez ici.

Une fois que Visual Studio est installé, créez un nouveau projet en cliquant sur **Fichier** > **Nouveau** > **Projet**, puis choisissez **Application console**.

Si tout va bien, vous devez obtenir cela :

```
01 using System;
02 using System;
03 using System.Collections.Generic;
04 using System.Linq;
05 using System.Text;
06
07 namespace ConsoleApplication1
08 {
09     class Program
10     {
11         static void Main(string[] args)
12         {
13         }
14     }
15 }
```

Si c'est le cas, alors tout va bien, nous pouvons commencer !

2 Notions basiques

2.1 Contexte

Pour illustrer les notions de la POO que nous allons vous enseigner, nous utiliserons tout au long de ce tutoriel l'exemple d'une gestion de banque.

Dans cette banque, les clients posséderont au minimum un compte bancaire (ils pourront en avoir plusieurs). Ce compte sera soit de type Courant, soit de type Epargne.

2.2 Classe et objet

Notion fondamentale et totalement indispensable à la POO, la **classe**.

Une classe correspond à la définition d'un ensemble d'entités partageant les mêmes attributs, les mêmes opérations et la même sémantique. En fait, une classe sert à modéliser un élément et va servir en tant que conteneur.

Dans notre cas, nous devons modéliser des clients qui posséderont des comptes bancaires. Commençons donc par créer une classe *Client*.

Pour cela, sous Visual Studio, dans le panneau droit **Solution Explorer**, cliquez droit sur le nom de votre projet (ConsoleApplication par défaut), puis cliquez sur **Ajouter** > **Classe**. Appelez votre classe *Client*, puis confirmez.

Visual Studio va alors vous créer votre première classe grâce au mot clé **class**.

```
1 using System;
2 class Client
3 {
4     // Contenu de la classe
5 }
```

Pour l'instant, il ne s'agit que d'une définition, votre classe est inutilisable en tant que telle. Pour l'utiliser, il faut tout d'abord comprendre la notion d'**objet**.

Un objet, caractérisé par un nom, correspond à l'instance d'une classe. Un objet regroupe un ensemble d'informations qui correspondent en fait aux valeurs des attributs d'une même classe.

On peut alors créer un objet de type Client nommé **monPremierClient**, puis un autre nommé **monSecondClient**, etc. Ces objets ainsi nommés permettront d'agir directement sur le contenu de la classe Client.

```
static void Main(string[] args)
{
     Client monPremierClient = new Client();
     Client monSecondClient = new Client();
     Client monTroisiemeClient = new Client();
}
```

Ces quelques lignes de codes, à placer dans le fichier Program.cs (qui contient la méthode **Main** et qui servira de point de démarrage), servent à instancier trois objets de type Client.

Pour instancier un objet, la syntaxe à respecter est donc la suivante :

Nom_De_La_Classe *identifiant* = new **Nom_De_La_Classe**();

La deuxième partie, new Client(), correspond en fait à l'appel du constructeur. Vous verrez cette notion dans la partie 2.4.

Si ces deux notions vous paraissent pour l'instant très abstraites, c'est normal, ne vous inquiétez pas. N'hésitez pas à relire plusieurs fois les définitions, puis passez à la suite.

2.3 Attributs, méthodes et visibilité

Pour le moment, notre classe est créée, mais elle est vide. Pour modéliser nos clients à travers la classe du même nom, il va falloir ajouter à cette classe des **attributs**.

Un attribut correspond à une variable stockant une information au sein d'une classe. Les attributs sont en fait des caractéristiques de la classe dans laquelle ils sont contenus.

Dans notre cas, qu'est-ce que qui caractérise un client ? Son nom, son prénom, son âge, sa taille, son poids, son sexe, sa couleur de peau, etc. Nous allons donc créer ces attributs.

```
class Client
{
    // Attributs
    private string _nomClient;
    private string _prenomClient;
    private int _ageClient;
    private double _tailleClient;
}
```

Voici quelques détails sur les quatres attributs créés :

- _nomClient de type chaine de caractère
- _prenomClient de type chaine de caractère
- _ageClient de type entier
- _tailleClient de type double

Vous vous demandez probablement à quoi sert le mot "clé" **private** ? Il s'agit de la **visibilité** de nos attributs.

La visibilité sert à définir les droits d'accès aux données. Il existe pour cela trois niveaux correspondant à trois mots clés :

- **public** : Il s'agit du niveau le moins élevé. Un attribut définit comme public sera accessible depuis n'importe quel endroit.
- **private** : Un attribut définit en tant que **private** ne sera accessible qu'au sein de la classe dans laquelle il est définit. Il s'agit du plus haut niveau de protection.
- **protected** : Les attributs définis en tant que **protected** seront accessibles au sein de la classe dans laquelle ils sont définis, mais également au sein de toutes les classes dérivées. Cette notion sera expliquée plus tard, oubliez-la pour l'instant.

Par convention, les attributs de type **private** doivent commencer par un underscore "_" en C#. De même, tout attribut doit utiliser une notation dite lowerCamelCase, c'est-à-dire dont le premier mot commence par une minuscule et tous les autres par une majuscule.

Par défaut, en C#, les attributs et méthodes possède le niveau de protection **private**, donc accessible uniquement au sein de la classe dans laquelle ils sont définis.

En fait, il existe un quatrième niveau de protection nommé **internal**. Sachez simplement que celui-ci est utilisé par défaut sur les classes et permet, grossomodo, d'autoriser l'accès aux classes dans tout votre projet.

Passons maintenant aux **méthodes**. Il s'agit en fait de fonctions définies au sein de votre classe et qui vont vous permettre d'agir sur vos attributs. Tout comme les attributs justement, les méthodes possèdent un niveau de visibilité.

Voici un exemple :

```
1 public void HelloWorld()
2 {
3     Console.WriteLine("Hello World !");
4 }
```

Le premier élément (public) défini la visibilité de la méthode. Le deuxième correspond au type de retour, ici void puisqu'il n'y a aucun retour. Le troisième correspond au nom de la méthode et le quatrième à la signature de la méthode. Ici () puisqu'il n'y a aucun paramètre à passer.

"Console.WriteLine" sert tout simplement à afficher le texte passé en paramètre dans le flux de sortie standard, ici la console.

Puisque notre méthode est déclarée comme public, cela signifie qu'elle est accessible depuis n'importe où. Nous pouvons donc utiliser nos objets précédemment déclarés pour appeler cette méthode.

```
static void Main(string[] args)
{
    Client monPremierClient = new Client();
    monPremierClient.HelloWorld();
}
```

Comme vous le voyez, c'est très simple. Il suffit d'utiliser votre objet, puis de faire suivre celui-ci du nom de la méthode séparée d'un point. D'ailleurs, en mettant le point, Visual Studio vous proposera automatiquement un ensemble d'attributs et méthodes accessibles.

Vous pouvez dorénavant compiler votre projet et lancer l'application. Pour cela, cliquez sur **Débugger** > **Lancer sans débugger** ou utilisez le raccourci **Ctrl + F5**. Si vous avez suivi le tutoriel correctement, alors vous devriez voir apparaître le texte "Hello World !" affiché dans la console.

Comme vous l'avez vu, cette méthode n'utilise pas les attributs de notre classe Client, elle se contente simplement d'afficher du texte. En effet, nous ne pouvons pas encore utiliser les attributs puisque ceux-ci ne possèdent pour l'instant aucune valeur.

2.4 Constructeur

Lorsque nous avons créé notre premier objet tout à l'heure, je vous ai parlé de constructeur. Il s'agit en fait d'une méthode spéciale qui sera appelée lors de l'instanciation d'un objet et qui servira à initialiser les valeurs de vos attributs.

Voici un premier exemple :

```
class Client
{
    // Attributs

    // Constructeur
    public Client()
    {
        this._nomClient = "";
        this._prenomClient = null;
        this._ageClient = 0;
        this._tailleClient = 0.0;
    }
}
```

Sa définition est la même que pour n'importe quelle autre méthode à ceci près que le constructeur ne nécessite pas de spécifier un type de retour. Son nom doit obligatoirement correspondre au nom de votre classe.

Dans notre constructeur, nous initialisons les valeurs de chacun des attributs caractérisant notre classe *Client*. Comme vous le voyez, vous pouvez utiliser des chaines de caractère, des entiers et même la valeur null.

Dans cet exemple, j'ai également introduit le mot-clé **this**. Ce dernier fait référence à l'objet à partir duquel il est appelé. Ainsi, en appelant ce constructeur depuis votre objet monPremierClient, **this** fera référence à cet objet et initialisera les valeurs afin qu'elle soit disponible uniquement pour l'instance de la classe *Client* en cours d'utilisation.

Imaginons maintenant que nous récupérons des clients stockés dans une base de données. Nous voulons instancier un objet pour chaque client et donner directement aux attributs les valeurs récupérées dans la base.

C'est tout à fait possible ! Il suffit simplement de spécifier un nouveau constructeur. En fait, vous pouvez créer autant que constructeur que vous le souhaitez tant que leur signature est différente.

```
public Client(string nom, string prenom, int age, double taille)
{
    this._nomClient = nom;
    this._prenomClient = prenom;
    this._ageClient = age;
    this._tailleClient = taille;
}
```

Nous voici donc avec un nouveau constructeur qui prend cette fois-ci en paramètre quatre valeurs afin d'initialiser nos attributs.

L'appel d'un constructeur avec des paramètres se fait de la même manière qu'un constructeur sans paramètre, à la différence qu'il faut lui passer des valeurs.

```
static void Main(string[] args)
{
    Client monClient = new Client("Skywalker","Luke",30,1.80);
}
```

Dans l'exemple ci-dessus, nous créons un objet de type *Client* nommé *monClient*. Les attributs seront initialisés de la façon suivante :

- _nomClient : "Skywalker"
- _prenomClient : "Luke"
- _ageClient : 30
- _tailleClient : 1.80

Maintenant que nos attributs possèdent tous une valeur, nous devrions pouvoir accéder aux données n'est-ce pas ? Hé bien non, toujours pas !

Rappelez-vous, nous avons défini nos attributs avec le niveau de visibilité **private** ce qui les rend inaccessibles en dehors de la classe dans laquelle ils sont définis.

2.5 Accesseur et mutateur

Pour rendre nos attributs accessibles en dehors de la classe, il va falloir créer ce que l'on appelle des **accesseurs** (ou getter) et des **mutateurs** (ou setter). Là encore, il ne s'agit ni plus ni moins que de méthodes dont le but est d'accéder aux données.

Les accesseurs serviront à accéder aux données en lecture, c'est-à-dire à récupérer les données, tandis que les mutateurs permettront d'y accéder en écriture, c'est-à-dire de modifier les valeurs des attributs. Pour le reste de ce tutoriel, j'utiliserais les mots **getter** et **setter** à la place d'accesseur et mutateurs, ces derniers étant très peu utilisés.

Commençons avec un exemple pour accéder au nom de notre client.

```
// Accesseur / Getter
public string GetNomClient()
{
    return this._nomClient;
}
```

Pour que notre getter soit accessible n'importe où, nous lui donnons une visibilité **public**. L'attribut étant de type chaîne de caractère (string), nous définissons le type string en tant que valeur de retour pour notre getter.

Le nom de votre getter importe peu mais, par convention, ils sont souvent appelés Get+Nom_attribut en UpperCamelCase, c'est-à-dire avec une majuscule en début de chaque mot.

Le contenu de la fonction est très simple, il s'agit simplement de retourner la valeur de l'attribut _nomClient faisant référence à l'objet en cours d'utilisation (this).

Ainsi, bien que votre attribut soit privé, vous pourrez dorénavant accéder à sa valeur depuis n'importe quel endroit grâce au getter publique que vous venez de créer.

Le principe du setter est exactement le même, sauf qu'il s'agit cette fois de modifier les données.

```
// Mutateur / Setter
public void SetNomClient(string nom)
{
   this._nomClient = nom;
}
```

Les setter n'ayant aucune nécessité de retourner une valeur, ceux-ci sont définis avec **void** en type de retour. Par convention, ils seront nommés Set+Nom_attribut eux aussi en UpperCamelCase, tout comme toute autre méthode d'ailleurs.

Le contenu est très simple également puisqu'il s'agit simplement de l'affectation d'une valeur à notre attribut. Vous pourrez dorénavant modifier les valeurs de vos attributs à partir de n'importe où.

Mettons tout de suite ceci en pratique.

```
static void Main(string[] args)
{
   Client monClient = new Client();
   monClient.SetNomClient("Toto");
   Console.WriteLine("Mon client s'appelle {0}",
monClient.GetNomClient());
}
```

Dans cet exemple :

1. Nous instancions un objet de type *Client* nommé monClient grâce au constructeur sans paramètre.
2. Nous utilisons le setter nommé SetNomClient afin de donner la valeur "Toto" à l'attribut _nomClient.
3. Nous affichons dans la console "Mon Client s'appelle Toto" grâce à l'appel du getter GetNomClient retournant la valeur de l'attribut _nomClient.

Pour vous entraîner, vous pouvez définir les setter et getter relatif à chaque attribut puis créer, par exemple, une méthode ObtenirInformation() qui écrirait dans la console l'ensemble des informations connues sur le client.

2.6 Propriété et propriété automatique

Les **propriétés** sont utilisées pour rendre accessible les getter et setter via un nom commun. Prenons directement un exemple en créant une propriété pour _prenomClient.

```
// Propriété
public string PrenomClient
{
    get
    {
        return this._prenomClient;
    }
    set
    {
        this._prenomClient = value;
    }
}
```

Notre propriété, équivalente à un getter et setter combiné, doit prendre comme type de retour le type de l'attribut concerné. Ici, il s'agit de _prenomClient, soit d'une chaine de caractère, la méthode doit donc prendre un type string en tant que type de retour.

Le principe est très simple. Au lieu de posséder deux méthodes (getter et setter) séparées, les deux sont regroupées dans notre propriété au sein de deux blocs **get** et **set**. Cette dernière peut être appelée comme une simple méthode et Visual Studio décidera automatiquement si il doit appeler le getter ou le setter.

```
1 static void Main(string[] args)
2 {
3     Client monClient = new Client();
4     monClient.PrenomClient = "Toto";
5     string prenomClient = monClient.PrenomClient;
6     Console.WriteLine(prenomClient);
7 }
```

Première ligne, je pense que vous avez maintenant saisi le principe, nous instancions un objet de type *Client* et nommé monClient.

A la deuxième ligne, nous utilisons notre propriété PrenomClient en tant que setter et attribuons par le fait la valeur "Toto" à l'attribut _prenomClient.

Troisième ligne, nous utilisons la propriété PrenomClient en tant que getter et récupérons la valeur contenu dans l'attribut _prenom afin de la stocker dans un string nommé prenomClient. Nous affichons ensuite ce prenom dans la console.

La gestion des getter et setter devient ainsi transparent pour le développeur qui n'a plus qu'à retenir le seul nom de la propriété.

Encore plus simple, il existe les **propriétés automatiques**. Ces dernières combinent à la fois l'attribut, son setter et son getter.

Dans l'exemple suivant, nous "transformons" notre attribut _ageClient en propriété automatique et mettons à jour les constructeurs pour prendre en compte cette modification.

```
01 class Client
02 {
03     // Attributs
04     private string _nomClient;
05     private string _prenomClient;
06     private double _tailleClient;
07
08     // Propriété
09     public int AgeClient { get; set; }
10
11     // Premier Constructeur
12     public Client()
13     {
14         this._nomClient = "";
15         this._prenomClient = null;
16         this.AgeClient = 0;
17         this._tailleClient = 0.0;
18     }
19
20     // Second constructeur
21     public Client(string nom, string prenom, int age, double taille)
22     {
23         this._nomClient = nom;
24         this._prenomClient = prenom;
25         this.AgeClient = age;
26         this._tailleClient = taille;
27     }
28 }
```

Comme vous le voyez, une simple ligne nous permet de définir notre propriété. Nous obtenons donc :

- L'équivalent d'un attribut nommé AgeClient de type entier
- Un getter
- Un setter

Son appel se fait comme une simple propriété.

Il est certes bien plus rapide d'utiliser une propriété que de créer un attribut, un setter et un getter, mais il est tout aussi important de comprendre le fonctionnement de ces trois éléments lorsqu'ils sont déclarés séparément.

2.7 Récapitulatif

Cette première partie a déjà probablement due vous donner du fil à retordre. Néanmoins, il s'agit là de notions fondamentales qu'il faut absolument comprendre et maîtriser afin de passer à la suite.

Aussi, je vous je vous encourage à vous entraîner à compléter la classe Client pour comprendre parfaitement ces premiers principe.

Voici un exemple de ce à quoi votre classe complète pourrait ressembler.

```
01 class Client
02 {
03     // Attributs
04     private string _nomClient;
05     private string _prenomClient;
06
07     // Propriétés automatiques
08     public int AgeClient { get; set; }
09     public double TailleClient { get; set;}
10
11     // Constructeur
12     public Client()
13     {
14         this._nomClient = "";
15         this.PrenomClient = null;
16         this.AgeClient = 0;
17         this.TailleClient = 0.0;
18     }
19
20     // Deuxième constructeur
21     public Client(string nom, string prenom, int age, double taille)
22     {
23         this._nomClient = nom;
24         this.PrenomClient = prenom;
25         this.AgeClient = age;
26         this.TailleClient = taille;
27     }
28
29     // Accesseur / Getter
30     public string GetNomClient()
31     {
32         return this._nomClient;
33     }
34
35     // Mutateur / Setter
36     public void SetNomClient(string nom)
37     {
38         this._nomClient = nom;
39     }
40
41     // Propriété
42     public string PrenomClient
43     {
44         get
45         {
46             return this._prenomClient;
47         }
48         set
49         {
50             this._prenomClient = value;
51         }
52     }
53
54     // Méthodes
55     public void HelloWorld()
56     {
57         Console.WriteLine("Hello World ! ");
58     }
59
60     public void ObtenirInformation()
61     {
62         Console.WriteLine("Ce client s'appelle {0} {1}, a {2} ans et
mesure {3}mètre.", this.PrenomClient, this._nomClient,
this.AgeClient, this.TailleClient);
63     }
64 }
```

Voici maintenant un exemple d'utilisation.

```
01 static void Main(string[] args)
02 {
03     // Création d'un client grâce au constructeur vide
04     // Utilisation des setter
05     Client monPremierClient = new Client();
06     monPremierClient.SetNomClient("John");
07     monPremierClient.PrenomClient = "Martin";
08     monPremierClient.AgeClient = 20;
09     monPremierClient.TailleClient = 1.75;
10
11     // Création d'un client grâce au constructeurs avec paramètres
12     Client monSecondClient = new Client("Skywalker", "Luke", 30,
   1.80);
13
14     // Création d'un client grâce aux valeurs saisies par
   l'utilisateur
15     Console.Write("Nom :");
16     string nom = Console.ReadLine();
17     Console.Write("Prenom :");
18     string prenom = Console.ReadLine();
19     Console.Write("Age :");
20     int age = int.Parse(Console.ReadLine());
21     Console.Write("Taille :");
22     double taille = double.Parse(Console.ReadLine());
23
24     Client monTroisiemeClient = new Client(nom, prenom, age, taille);
25
26     // Affichage des informations
27     monPremierClient.ObtenirInformation();
28     monSecondClient.ObtenirInformation();
29     monTroisiemeClient.ObtenirInformation();
30 }
```

Pour clore cette première partie, faisons un rapide rappel de tout ce que nous venons de voir.

- Une classe correspond à la définition d'un ensemble d'entités partageant les mêmes attributs et méthodes.
- Un objet correspond à l'instanciation d'une classe.
- Chaque attribut et méthode possède un niveau de visibilité qui peut être soit **public**, c'est-à-dire accessible depuis n'importe où, soit **private**, c'est-à-dire accessible depuis la classe uniquement, ou soit **protected**, c'est-à-dire accessible depuis la classe où il est défini et depuis les classes dérivées (notion que nous verrons dans la partie suivante).
- Pour que les attributs prennent des valeurs lorsque l'on instancie un objet, il faut utiliser les constructeurs. Une classe peut contenir un nombre infini de constructeur tant que ceux-ci ne possède pas la même signature.
- Pour accéder aux attributs d'une classe, il faut utiliser des getter (lecture) et setter (écriture).
- Une propriété regroupe à la fois un getter et un setter relatif à un même attribut.
- Une propriété automatique regroupe l'attribut, son getter et son setter.

Dans la partie suivante, vous apprendrez entre autre ce qu'est l'héritage, le mot clé static, les interfaces, les classes abstraites et les classes imbriquées.

3 Notions basiques, suite

3.1 L'héritage

Après avoir vu la création des objets, l'initialisation des données (constructeur) et l'accès aux données (getter/setter), nous allons nous attaquer à des notions un peu plus évoluées. Néanmoins, il s'agit toujours de notions fondamentales que vous devez absolument connaître pour maîtriser la programmation orientée objet.

Rappelez-vous, dans notre cas de gestion de banque, chaque client doit posséder au minimum un compte. Ce compte contient un identifiant, un libelle, un solde et une date d'ouverture.

Pour la suite de cette partie, vous allez devoir créer la classe Compte. Les fainéants peuvent directement la récupérer ci-dessous.

```
01 class Compte
02 {
03     // Attributs
04     public int IdCompte { get; set; }
05     public string LibelleCompte { get; set; }
06     public double SoldeCompte { get; set; }
07     public DateTime DateOuvertureCompte { get; set; }
08 
09     // Constructeurs
10     public Compte()
11     {
12         this.IdCompte = 0;
13         this.LibelleCompte = "";
14         this.SoldeCompte = 0.0;
15         this.DateOuvertureCompte = new DateTime();
16     }
17 
18     public Compte(int id, string libelle, double solde, DateTime
   date)
19     {
20         this.IdCompte = id;
21         this.LibelleCompte = libelle;
22         this.SoldeCompte = solde;
23         this.DateOuvertureCompte = date;
24     }
25 
26     // Méthode
27     public void ObtenirInformationCompte()
28     {
29         Console.WriteLine("Le compte {0} ouvert le {1} est crédité de
   {2}", this.LibelleCompte, this.DateOuvertureCompte,
   this.SoldeCompte);
30     }
31 }
```

Si vous ne comprenez pas quelque chose dans cette classe, alors vous avez probablement mal compris une notion expliquée dans la partie précédente. La seule nouveauté est l'utilisation d'un attribut de type DateTime qui sert, comme son nom l'indique, à stocker une valeur

relative à une date. Vous pouvez utiliser cette attribut en tant que string si vous souhaitez simplifier son utilisation.

Pour notre exemple, je vous ai également dis qu'un compte devait être soit de type courant, soit de type épargne. Alors, faut-il créer deux autres classes quasiment identique à celle que nous venons de créer ? Non ! C'est ici qu'intervient l'**héritage**.

L'héritage permet à une classe d'hériter des données et méthodes d'une autre classe. On obtiendra alors un système classe fille/mère (ou encore classe dérivée/générique) qui nous permettra de posséder plusieurs classes possédant plus ou moins la même définition, sans pour autant écrire deux fois cette même classe.

Le principal intérêt de l'héritage est de pouvoir ajouter des attributs et méthodes à une classe tout en profitant de ceux de la classe mère. C'est exactement ce qu'il nous faut puisque chaque type de compte possède un identifiant, un libelle, un solde et une date d'ouverture, mais un compte épargne doit également posséder un taux d'intérêt et un plafond, alors que le compte courant possède un découvert autorisé.

Sachez également qu'une classe dérivée ne peut pas posséder un niveau de visibilité plus élevé que sa classe générique.

Voyons tout de suite comment créer une classe dérivée.

```
1 class CompteCourant : Compte
2 {
3     // Définition de la classe
4 }
```

Enfantin non ? Il suffit simplement de faire suivre le nom de la classe d'un double point, puis du nom de la classe générique. De ce simple fait, notre classe *CompteCourant* bénéficiera des attributs et méthodes de la classe *Compte*.

La programmation objet introduit également l'héritage multiple. Il s'agit simplement du fait qu'une classe puisse posséder plusieurs classes génériques. En C#, cette notion n'est pas disponible, du moins pas avec le framework 3.5, actuellement standard.

Voici notre classe CompteCourant, entièrement créée.


```
01 class CompteCourant : Compte
02 {
03     // Propriété
04     public double DecouvertAutorise { get; set; }
05
06     // Constructeurs
07     public CompteCourant()
08         : base()
09     {
10         this.DecouvertAutorise = 0;
11     }
12
13     public CompteCourant(int id, string libelle, double solde,
   DateTime date, double decouvert)
14         : base(id, libelle, solde, date)
15     {
16         this.DecouvertAutorise = decouvert;
17     }
18 }
```

Au niveau des propriétés, comme vous le voyez, seul *DecouvertAutorise* est créée. Notre classe CompteCourant étant dérivée de Compte, celle-ci bénéficie directement des attributs IdCompte, LibelleCompte, SoldeCompte, DateOuvertureCompte de sa classe mère Compte.

La nouveauté, c'est au niveau des constructeurs avec l'introduction du mot-clé **base**. Celui-ci fait référence au constructeur de la classe générique par rapport à la classe dans laquelle il est appelé. Ici, il appellera le constructeur sans paramètre (Compte()) de la classe *Compte*.

La syntaxe est la même que pour spécifier l'héritage entre deux classes, il suffit de créer le constructeur CompteCourant() comme si de rien n'était, puis de faire suivre ce dernier par base() précédé d'un double point. De ce fait, les attributs de la classe Compte seront initialisés, ainsi que l'attribut DecouvertAutorise, propre à la classe CompteCourant.

Pour le deuxième constructeur, il s'agit du même principe sauf que, cette fois, nous passons des paramètres au constructeur. Pour pouvoir passer des paramètres au constructeur base(), il faut absolument que ces derniers soient définis dans la signature du constructeur fille, soit ici CompteCourant().

De ce fait, les attributs de la classe Compte seront initialisés avec les valeurs que vous aurez passées en paramètre, tout comme pour l'attribut DecouvertAutorise, propre à CompteCourant.

Si vous avez compris le principe, vous devriez être capable de réaliser la classe CompteEpargne. Celle-ci possède, en plus de tous les attributs d'un compte standard, un taux d'intérêt, ainsi qu'un plafond.

```
class CompteEpargne : Compte
{
    // Attributs
    public double Interet { get; set; }
    public int Plafond { get; set; }

    // Constructeurs
    public CompteEpargne()
        : base()
    {
        this.Interet = 0;
        this.Plafond = 0;
    }

    public CompteEpargne(int id, string libelle, double solde,
DateTime date, double interet, int plafond)
        : base(id, libelle, solde, date)
    {
        this.Interet = interet;
        this.Plafond = plafond;
    }
}
```

Le principe est exactement le même et vous devriez normalement être capable de comprendre le code ci-dessus tout seul.

Pour utiliser des classes héritées, le principe est le même qu'avec les classes standards. Voici un exemple ci-dessous.

```
Compte monCompte = new Compte();
monCompte.IdCompte = 1;
monCompte.LibelleCompte = "MonPremierCompte";
monCompte.DateOuvertureCompte = new DateTime(2007, 04, 21);
monCompte.SoldeCompte = 50000.45;
monCompte.ObtenirInformationCompte();

CompteCourant cc = new CompteCourant(2,"SecondCompte",10000,new
DateTime(2010,02,27),1000);
cc.ObtenirInformationCompte();

CompteEpargne ce = new CompteEpargne();
ce.IdCompte = 3;
ce.LibelleCompte = "TroisiemeCompte";
ce.DateOuvertureCompte = new DateTime(1972,01,01);
ce.SoldeCompte = 1000000;
ce.Interet = 1;
ce.Plafond = 1500000;
ce.ObtenirInformationCompte();
```

Dans un premier temps, nous instancions un objet monCompte de type Compte, tout ce qu'il y a de plus normal. En deuxième, nous instancions un objet cc de type CompteCourant en appelant le constructeur contenant des paramètres. Souvenez-vous, ce dernier va seulement garder la valeur relative à l'attribut DecouvertAutorise (ici, 1000) puis passer les autres au constructeur de la classe mère Compte.

Troisième exemple avec un objet de type CompteEpargne. Cette fois-ci nous utilisons le constructeur vide (qui va attribuer des valeurs par défaut à nos attributs) puis appelons les méthodes de la classe mère Compte pour donner des valeurs à nos attributs.

Comment se fait-il que l'appel de ces méthodes soit possible ? Car ces dernières sont définies avec une visibilité **public**, donc accessible de n'importe où.

Mais dans ce cas, comment faire pour que celles-ci ne soient pas accessibles depuis n'importe quel endroit, mais que nous y ayons toujours accès depuis notre classe dérivée ?

Dans la première partie, je vous parlais d'un niveau de visibilité **protected**. C'est ici que celui-ci intervient. Ce niveau de visibilité permet aux attributs et méthodes qui l'utilisent d'être accessibles uniquement depuis la classe dans laquelle ils sont définis et dans les classes dérivées.

Voyons ce qu'il se passe si nous définissons le getter IdCompte en tant que protected.

```
1 public int IdCompte { protected get; set; }
```

Vous pouvez dorénavant essayer de récupérer la valeur de IdCompte depuis votre fichier Program.cs, cela sera impossible. A partir de maintenant, l'attribut IdCompte n'est accessible que dans la classe Compte et dans les classes qui en dérivent.

```
1 // Retourne une erreur
2 // Getter inaccessible
3 int id = monCompte.IdCompte;
```

En revanche, nous pouvons par exemple créer une méthode recupererIdCompte() au sein de la classe CompteEpargne qui, elle, nous autorisera à récupérer la valeur de cet attribut.

```
1 // Dans CompteEpargne.cs
2 public int recupererIdCompte()
3 {
4     return this.IdCompte;
5 }
```

```
1 // Dans Program.cs
2 // Ne génère pas d'erreur
3 int id = ce.recupererIdCompte();
```

Dans l'exemple ci-dessus, la méthode recupererIdCompte, déclarée avec une visiblité **public**, nous permet d'accéder à la valeur de l'attribut IdCompte, pourtant déclaré en tant que **protected**.

Pour la suite de ce tutoriel, nous considérerons que le getter IdCompte est à nouveau public.

3.2 Les classes scellées

Le principe des classes scellées en C# est très simple, il s'agit simplement de réduire les fonctionnalités d'héritage. Une classe scellé, définie avec le mot clé **sealed** est une classe que vous ne pouvez pas dériver.

Ainsi, si nous définissons notre classe Compte en tant que classe scellée, il sera impossible de créer nos classes ComtpeEpargne et compteCourant en tant que classes dérivées de Compte.

```
// Classe Compte scellée
sealed class Compte
{
    // Attributs, constructeurs, méthodes,...
}
```

```
// Classe CompteEpargne dérivée de Compte
// Provoque une erreur lors de la compilation
// Impossible d'hériter d'une classe scellé
class CompteEpargne : Compte
{
    // Attributs, constructeurs, méthodes,...
}
```

Pour la suite de ce tutoriel, nous repassons la classe Compte en tant que classe non scellée.

3.3 Le mot clé static

Voici une nouvelle notion fondamentale à la POO, l'utilisations du mot-clé **static**. Celui-ci peut s'appliquer aussi bien à une classe, qu'à un attribut ou une méthode. Nous traiterons ces trois cas séparément.

Une méthode, attribut ou propriété déclarée avec le mot clé **static** peut être accessible sur une classe même si aucun n'objet de cette classe n'a été instancié.

Dans notre cas, imaginons que le taux d'intérêt de nos comptes épargne soit fixé à 1.2%. Plutôt que d'initialiser ce taux à 1.2 avec le constructeur, nous allons modifier l'attribut pour le définir en tant que static. Ainsi, la valeur sera accessible à tout moment sans instancier d'objet de type CompteEpargne.

```
public static double _interet = 1.2;
```

```
// Dans Program.cs
Console.WriteLine(CompteEpargne._interet.ToString());
```

Comme vous le constatez, l'attribut static _interet est accessible alors que nous n'avons instancié aucun objet.

Le principe est le même pour les méthodes statiques. Ces dernières sont accessibles sans instancier d'objet. Une méthode statique ne peut pas permettre d'accéder à des éléments non statiques.

Voici un exemple d'utilisation.

```
public static void MethodeStatique()
{
     Console.WriteLine("Cette méthode est statique");
}
```

```
// Dans Program.cs
CompteEpargne.MethodeStatique();
```

La méthode, définie en tant que méthode statique peut être appelée sans instancier d'objet. En revanche, l'exemple suivant ne fonctionnera pas :

```
// Provoque une erreur lors de la compilation
// Impossible d'accéder à une donné non statique dans une méthode statique
public static void MethodeStatique2()
{
    Console.WriteLine(this.Plafond.ToString());
}
```

Enfin, les classes statiques :

- Elles ne peuvent pas être instanciées
- Elles doivent contenir uniquement des méthodes et attributs statiques
- Elles sont scellées par défaut (ne peuvent pas être dérivées)
- Elles ne peuvent pas contenir de constructeur

Voici un exemple d'utilisation de classe statique :

```
01 static class InformationBanque
02 {
03     public static string _deviseBanque = "Un pour tous et tous pour
   l'argent !";
04     private static string _couleurLogo = "bleu";
05
06     public static void EmpruntMaximum()
07     {
08         Console.WriteLine("Une banque peut vous prêter 10 000 euros
   au maximum.");
09     }
10
11     public static int RecupererAnneeCreationBanque()
12     {
13         return 1935;
14     }
15 }
```

```
1 // Dans Program.cs
2 InformationBanque.EmpruntMaximum();
3 int anneeCreation = InformationBanque.RecupererAnneeCreationBanque();
4 Console.WriteLine(anneeCreation.ToString());
```

Notre classe *InformationBanque* contient un ensemble d'informations relatives à la banque et qui, par conséquent ne changeront jamais, c'est pour cela qu'il est judicieux de définir cette classe en tant que statique.

InformationBanque contient uniquement des méthodes et attributs statiques. Le contraire engendrerait un problème lors de la compilation.

Ce qu'il faut retenir :

- Une classe statique ne contient que des éléments statiques, mais des éléments statiques ne sont pas forcément contenus dans une classe statique.
- Les méthodes et attributs définis comme statiques sont accessibles sans instancier d'objet.

3.4 Les classes abstraites

Dans la partie 3.2, nous vous apprenions ce qu'était une classe scellée, c'est-à-dire une classe qui ne peut pas être dérivée. Les **classes abstraites** sont en quelques sortes l'inverse des classes scellées, il s'agit de classe qui doivent être dérivées pour être utilisable.

En effet, une classe déclarée en tant que classe abstraite n'est pas instanciable. En revanche, contrairement à une classe static, elle n'est pas accessible, ce qui la rend inutilisable en tant que telle.

Pour revenir à notre exemple, si l'on réfléchit bien au contexte, un compte est soit un compte épargne, soit un compte courant, mais ne peut pas être un compte tout simplement. Dans ce cas, il serait logique de définir notre classe *Compte* en tant que classe abstraite afin que cette dernière ne puisse pas être instanciée.

```
abstract class Compte
{
    // Attributs, méthodes, ...
}
```

Il suffit simplement de rajouter le mot **abstract** devant le mot-clé **class**. Cette simple modification nous empêche d'instancier tout objet de type *Compte*. Vous pouvez essayer de le faire via l'exemple utilisé précédemment, vous obtiendrez une erreur lors de la compilation.

En revanche, cela ne change absolument rien pour nos classes *CompteEpargne* et *CompteCourant*. Ces deux classes dérivent toujours de la classe *Compte* et peuvent toujours être instanciées et utilisables.

Sachez qu'il est également possible de définir des méthodes abstraites. Celles-ci doivent alors :

- Etre obligatoirement contenues dans une classe abstraite.
- Ne contenir qu'une signature et aucune définition.
- Etre obligatoirement implémentées dans toute classe dérivée.

Ceci faisant appel à un concept avancé de la POO, nous n'en parlerons pas plus pour l'instant. Vous retrouverez cette notion expliquée plus en détail dans les parties suivantes.

3.5 Les interfaces

Les **interfaces** sont très proches des classes abstraites.

- Une interface définit un contrat que la classe qui l'implémente doit respecter. Les interfaces ne peuvent pas être instanciées.
- Au lieu de parler de dérivation, on parle d'implémentation.
- Une interface ne possède que des méthodes. (En fait, elle peut posséder également d'autres éléments que nous verrons dans la partie avancée de ce tutoriel).
- Lorsqu'une classe implémente une interface, cette classe doit obligatoirement implémenter toutes ses méthodes.
- Les méthodes déclarées dans les interfaces ne possèdent pas de définition.

La déclaration d'une interface se fait grâce au mot-clé **interface**.

```
public interface IComparable
{
    void CompareTo(object obj);
}
```

Par convention, le nom d'une interface doit toujours commencé par "I". Dans notre exemple, IComparable est une interface propre au framework .NET, vous n'avez donc pas à la redéfinir.

L'interface comporte une unique méthode, CompareTo, qui prend en paramètre un objet. Admettons que notre classe Compte ait besoin de cette interface. Cette dernière devra alors implémenter la méthode CompareTo, sous peine de rencontrer une erreur lors de la compilation.

```
abstract class Compte : IComparable
{
    // Attributs, propriétés, méthodes,...

    public int CompareTo(object obj)
    {
        throw new NotImplementedException();
    }
}
```

L'implémentation d'une interface se fait de la même façon que l'héritage d'une classe générique, il suffit de séparer le nom des deux classes par un double-point.

Contrairement à l'héritage, une classe peut implémenter plusieurs interfaces. De même, une classe peut hériter d'une autre classe tout en implémentant une ou plusieurs interfaces. Pour cela, il suffit de séparer les classes génériques et interfaces par des virgules.

```
// Dérive de la classe Compte
// Implémente les interfaces IComparable et ICloneable
class CompteEpargne : Compte, IComparable, ICloneable
{
    // Attributs, méthodes,...
    // Implémentation des méthodes de IComparable
    // Implémentation des méthodes de ICloneable
}
```

Dans notre exemple, la classe *CompteEpargne* hérite de la classe *Compte*, mais implémente également les interfaces IComparable et ICloneable.

3.6 Les classes imbriquées

Jusqu'à maintenant, nous savions qu'une classe pouvait contenir des attributs, des propriétés et des méthodes. Mais ce n'est pas tout, les classes peuvent aussi contenir d'autres classes. Ce principe s'appelle **classe imbriquée**.

Les classes imbriquées sont souvent utilisées pour structurer une classe en différente sous classes qui n'auraient pas de réelles significations si elles étaient déclarées séparément. Cette notion permet également de cacher cette classe aux utilisateurs.

Seule une classe imbriquée peut être définie avec un niveau de visibilité **private** ou **protected.**

Voici un exemple avec une classe *Bijoux* déclarée en tant que classe imbriquée dans la classe Client.

```
01 class Client
02 {
03     public class Bijoux
04     {
05         public int NombreDeColliers { get; set; }
06         public string MarqueCeinture { get; set; }
07 
08         public Bijoux()
09         {
10             this.NombreDeColliers = 0;
11             this.MarqueCeinture = "";
12         }
13 
14         public int RecupererNombreColliers()
15         {
16             return this.NombreDeColliers;
17         }
18     }
19 }
```

Définie comme telle, la classe Bijoux ne sera accessible que depuis la classe Client. Cette notion étant assez peu utilisée, nous ne détaillerons pas plus son utilisation.

3.7 Initialisateur d'objets

Pour clore cette partie, nous allons voir ce qu'est un initialisateur d'objet. Cette notion est très simple, il s'agit d'une façon d'initialiser les valeurs des attributs d'un objet lors de sa définition sans pour autant utiliser le constructeur.

Pour cela, reprenons la définition de notre classe Client.

```
1 public class Client
2 {
3     // Attributs
4     private string _nomClient;
5     private string _prenomClient;
6     public int AgeClient { get; set; }
7     public double TailleClient { get; set;}
8 }
```

Lors de la création d'un objet Client, au lieu d'utiliser le constructeur, nous pouvez utiliser l'initialisateur d'objet pour accéder aux attributs et propriétés accessibles. Ici, on pourra initialiser les valeurs des propriétés AgeClient et TailleClient, puisque ces dernières sont définies en tant que **public**.

```
1 Client c = new Client { AgeClient = 10, TailleClient = 1.50 };
```

Le bout de code ci-dessus est équivalent à :

```
1 Client c = new Client();
2 c.AgeClient = 10;
3 c.TailleClient = 1.50;
```

Pour utiliser l'initialisateur d'objet, il suffit simplement d'instancier son objet sans appeler le constructeur, puis de spécifier les différents attributs et/ou propriétés séparés par des virgule, le tout entre accolades.

3.8 Récapitulatif

Beaucoup de notions ont été vues dans cette deuxième parties dédiées aux notions basiques de la programmation orientée objet. Encore une fois, il est fort probable que vous ne les ayez pas toutes saisies du premier coup. Je vous encourage donc à les revoir afin de les maîtriser pour la suite.

Voici un petit récapitulatif des notions vues dans cette seconde partie :

- L'héritage permet à une classe de dériver d'une autre classe en lui permettant d'accéder à ses attributs et méthodes.
- Les attributs et méthodes déclarés avec un niveau de visibilité protected sont accessibles depuis la classe dans laquelle ils sont déclarés et dans les classes dérivés.
- Les classes scellées ne peuvent pas être dérivées.
- Les classes abstraites doivent être dérivées pour être utiles.
- Une classe abstraite peut contenir des méthodes non abstraites.
- Le mot clé static utilisé sur des attributs ou méthodes permet d'y accéder sans avoir besoin d'instancier d'objet.
- Une classe statique ne peut pas être instanciée, ne doit contenir que des éléments statiques et est scellée par défaut.
- Une interface définit un contrat que la classe qui l'implémente doit respecter.
- Les méthodes définies dans une interface ne doivent pas contenir de définition et doivent être obligatoirement redéfinis dans la classe qui implémente cette interface.
- Les classes imbriquées permettent de structurer des classes qui n'auraient aucune signification si elles étaient définies séparément.

- L'initialisateur d'objet permet d'initialiser les valeurs des attributs accessibles d'une classe, sans pour autant appeler de constructeur lorsque l'on instancie un objet.

Dans la prochaine partie, nous attaquerons les notions avancées. En vrac, nous pouvons citer les classes et méthodes partielles, les expressions lambda, la surcharge d'opérateur et de méthode, les types anonymes, les indexeurs, etc.

Quoi qu'il en soit, si vous maîtrisez déjà toutes les notions vues jusqu'à maintenant, vous êtes tout à fait capable de faire un programme correct. Les notions avancées sont, certes, très puissantes, mais pas indispensables.

[Pour l'instant, ce tutoriel s'arrête là. Il sera mis à jour très prochainement avec l'ajout des notions avancées]