1. Créer une classe

1.1.Présentation

Une **classe** est tout simplement un moule pour faire des objets.

Un objet est composé de **membres** ; parmi ces membres, on dispose de **champs** (les variables qui lui sont caractéristiques), de **méthodes**, de **propriétés**, ainsi que d'autres éléments que nous verrons plus tard. On a tendance à croire que "champ" et "membre" désignent la même chose alors que c'est faux : il faut bien voir qu'il existe plusieurs sortes de membres, dont les champs.

Prenez la bonne habitude de faire commencer le nom des champs par "m_", comme dans "m_age" (le 'm' n'étant pas sans lien avec "membre" ; en revanche cela ne concerne pas tous les membres mais juste les champs). C'est très important : cela permet de savoir quels champs vous avez créés.
C'est dans la classe que sont définis les membres (dont les champs et les méthodes). Tout objet créé à partir d'une classe possède les membres que propose cette classe.

Une classe simple se présente sous cette forme :

```
class nomDeLaClasse
{
   // Déclaration des champs

   // Déclaration des méthodes
}
```

Les champs sont de simples variables, vous savez donc les déclarer.

1.2.Le constructeur

C'est le nom que l'on donne à une méthode spéciale dans une classe. Le **constructeur** (c'est aussi un membre) d'une classe est appelé à chaque fois que vous voulez créer un objet à partir de cette classe. Vous pouvez donc écrire du code dans cette méthode et il sera exécuté à chaque création d'un nouvel objet.

Pour filer la métaphore du moule, les objets seraient les gâteaux que l'on peut faire avec et le constructeur serait en quelque sort notre cuisinier.

Le constructeur est la méthode qui a exactement le même nom que la classe, et qui ne retourne jamais rien. Il est donc inutile de préciser s'il retourne quelque chose ou non : on ne met même pas de void devant. Par contre, vous pouvez lui passer des arguments.
Lorsqu'il est appelé, le constructeur réserve un emplacement mémoire pour votre objet et si vous n'avez pas initialisé ses champs, il les initialise automatiquement à leur valeur par défaut.

Sachez aussi que vous n'êtes pas obligés d'écrire vous-mêmes le code du constructeur ; dans ce cas, un constructeur "par défaut" est utilisé. Si vous faites ainsi, lorsque l'objet est créé, tous ses champs qui ne sont pas déjà initialisés dans le code de la classe sont initialisés à leur valeur par défaut.

L'intérêt du constructeur est d'offrir au développeur la possibilité de personnaliser ce qui doit se passer au moment de la création d'un objet. Il rajoute en outre un aspect dynamique au code : vous pouvez affecter vos champs à l'aide de variables passées en paramètres.

Le destructeur

Le **destructeur** (c'est aussi un membre) est une méthode appelée lors de la destruction d'un objet. Son nom est celui de la classe, précédé d'un tilde '~'.
Là-aussi, rien ne vous oblige à mettre un destructeur. C'est seulement si vous voulez faire quelque chose de particulier à sa destruction. Cela peut notamment servir à libérer de la mémoire et à bien gérer les ressources ; c'est bien trop compliqué pour l'instant alors nous en parlerons en temps et en heure.

Exemple

Nous allons étudier une classe Person

```
public class Person
{
    private string m_name;
    public string Name
    {
        get { return m_name; }
        set { m_name = value; }
    }

    private ushort m_age;
    public ushort Age
    {
        get { return m_age; }
        set { m_age = value; }
    }

    public Person()
    {
        Console.WriteLine("Nouvelle personne créée.");
    }

    public Person(string name, ushort age)
    {
        this.m_age = age;
        this.m_name = name;
        Console.WriteLine("Nouvelle personne créée. Cette personne s'appelle " + name + " et a
" + age + " an(s).");
    }

    ~Person()
    {
        Console.WriteLine("Objet détruit.");
    }

    public void SayHi()
    {
        Console.WriteLine("Bonjour ! Je m'appelle " + this.m_name + " et j'ai " + this.m_age +
" ans.");
    }
```

```
}
```

Les **modificateurs** private et public se mettent devant **un type** (par exemple : une classe) ou **un membre** (par exemple : un champ ou une méthode).
private restreint l'accès de ce qui suit à l'usage exclusif dans le bloc où il a été déclaré.
public autorise quant à lui l'accès de ce qui suit depuis l'extérieur.
Le constructeur doit **impérativement** être précédé de public si vous voulez pouvoir l'appeler et créer un objet.
Par défaut, champs et méthodes utilisent le modificateur private, mais pour bien voir ce que l'on fait, il est préférable de toujours préciser. Nous verrons plus tard qu'il existe d'autres possibilités que public pour les classes elles-mêmes, mais ne nous y attardons pas pour l'instant.
Prenez comme habitude de définir vos champs en privé, pour éviter qu'ils puissent être modifiés de n'importe où et éviter des erreurs.
Vous pouvez cependant accéder publiquement à des champs privés, en ayant recours à des **propriétés**, comme dans cette classe avec la propriété Age :

```
public ushort Age
{
   get { return m_age; }
   set { m_age = value; }
}
```

À l'intérieur du bloc get, vous définissez comment se fait l'accès en **lecture**. Dans ce cas, si l'on veut récupérer la valeur de l'âge, on pourra écrire ushort userAge = toto.Age; (toto étant un objet de type Person).
À l'intérieur du bloc set, vous définissez comment se fait l'accès en **écriture**. Dans ce cas, si l'on veut changer la valeur de l'âge, on pourra écrire toto.Age = 10; (10 étant ici implicitement converti en ushort).
Que vient faire value dans tout ça ?

La variable value représente la valeur que l'on veut donner à m_age. Si l'on écrit toto.Age = 10;, value est un ushort qui vaut 10.
Cette variable est locale et n'existe que dans le bloc set.

Les propriétés ont un statut particulier ; en fait ce ne sont pas des variables mais des moyens d'accéder à des variables. D'ailleurs, get et set sont ce qu'on appelle des **accesseurs**.
Revenons à l'exemple de classe que je vous ai fourni. J'ai créé deux méthodes portant le même nom Person. Ce n'est pas une erreur, en fait j'ai **surchargé** le constructeur.
this est un mot-clef du langage C# qui désigne l'objet lui-même ("this" veut dire "ceci" en anglais). L'écriture this.m_age permet d'accéder au champ m_age de l'objet désigné par this.
this.m_age = age; aura pour effet d'initialiser l'âge de ma nouvelle personne avec l'entier que je passe comme paramètre au constructeur.
this est facultatif, donc par la suite j'écrirai m_age au lieu de this.m_age.

1.3.Créer un objet

Un objet est toujours issu d'une classe. On dit que c'est **une instance** de cette classe. La particularité des instances de classe est qu'elles se baladent toujours quelque part dans la mémoire, plus librement que les autres variables. Nous avons donc besoin d'une référence pour savoir où elles se trouvent et ainsi pouvoir les manipuler. Contrairement aux variables de type valeur qui contiennent une valeur que l'on manipule directement, les références ne font que

désigner une instance qui, elle, peut contenir une ou plusieurs valeurs. L'accès à ces valeurs se fait donc indirectement.

Pour manipuler une instance de cette classe, on va devoir faire deux choses :

1. **Déclarer une référence** qui servira à désigner l'instance ;
2. Créer l' instance, c'est-à-dire **instancier ma classe**.

1.3.1. Déclarer une référence

Une référence est avant tout une variable, et se déclare comme toutes les variables. Son type est le nom de la classe que l'on compte instancier :

```
Person toto;
```

On vient de déclarer une référence, appelée toto, qui est prête à désigner un objet de type Person.
Quelle est la valeur de cette référence ?

Nous n'avons fait que déclarer une référence sans préciser de valeur ; elle a dans ce cas été initialisée à sa valeur par défaut, qui est null pour les références. Ce code est donc équivalent à :

```
Person toto = null;
```

Lorsqu'une référence vaut null, cela signifie qu'elle ne désigne aucune instance. Elle est donc inutilisable. Si vous tentez de vous en servir, vous obtiendrez une erreur à la compilation :

```
Person toto;
// Erreur à la compilation: "Use of unassigned local variable 'toto'".
toto.SayHi();
```

Le compilateur vous indique que vous essayez d'utiliser une variable qui n'a pas été assignée.

En revanche, vous pouvez tout à fait déclarer un champ sans l'instancier : le constructeur de la classe se charge tout seul d'instancier à leur valeur par défaut tous les champs qui ne sont pas déjà instanciés. C'est pourquoi dans la classe Person ci-dessus, j'ai pu écrire private string m_name; sans écrire private string m_name = string.Empty; ou encore private string m_name = "";.

NB : La valeur par défaut de tout type numérique est 0, adapté au type (0.0 pour un float ou un double).
La valeur par défaut d'une chaîne de caractères (string) est null, et non pas la chaîne vide, qui est représentée par string.Empty ou encore "".
La valeur par défaut d'un caractère (char) est '\0'.
La valeur par défaut d'un objet de type référence est null.

1.3.2. Instancier une classe

Pour instancier la classe Person, et ainsi pouvoir initialiser notre référence avec une nouvelle instance, on utilise le mot-clef new :

```
// Déclaration de la référence.
Person toto;
// Instanciation de la classe.
```

```
toto = new Person();
```
Comme pour toute initialisation de variable, on peut fusionner ces deux lignes :

```
// Déclaration + instanciation
Person toto = new Person();
```
À ce stade nous avons créé une nouvelle instance de la classe Person, que nous pouvons manipuler grâce à la référence toto.
Lorsque l'opérateur new est utilisé, le **constructeur** de la classe est appelé. Nous avons vu qu'il pouvait y avoir plusieurs surcharges du constructeur dans une même classe, comme c'est le cas dans la classe Person. La version du constructeur appelée grâce à new est déterminée par les paramètres spécifiés entre les parenthèses.
Jusqu'à présent nous nous sommes contentés d'écrire new Person(), et nous avons ainsi utilisé implicitement le constructeur sans paramètre (on l'appelle **constructeur par défaut**).
La classe Person possède un autre constructeur qui nous permet de préciser le nom et l'âge de la personne. Profitons-en pour préciser que toto s'appelle Toto et a 10 ans, au moment de créer notre instance :

```
Person toto = new Person("Toto", 10);
```
Il était aussi possible de faire cela en plusieurs temps, comme ceci :

```
Person toto = new Person();
toto.Name = "Toto";
toto.Age = 10;
```

Si vous ne comprenez pas tout de suite la syntaxe des deux dernières lignes du bout de code précédent, c'est normal : nous n'avons pas encore vu comment *utiliser* des objets (ça ne saurait tarder).

Maintenant que nous savons créer des objets, voyons plus en détail comment nous en servir.

1.4.Utiliser un objet

1.4.1. Accéder aux membres d'un objet

Un fois l'objet créé, pour accéder à ses membres il suffit de faire suivre le nom de l'objet par un **point**.
toto.m_age n'est pas accessible car m_age est défini avec private. On peut en revanche accéder à la propriété publique toto.Age et à la méthode publique toto.SayHi.
Voyons ce que donne la méthode SayHi() de notre ami Toto :

```
Person toto = new Person("Toto", 10);

toto.SayHi();
```
Et nous voyons apparaître comme prévu :

```
Bonjour ! Je m'appelle Toto et j'ai 10 ans.
```

1.4.2. Les propriétés

Pourquoi utiliser des propriétés alors que l'on peut utiliser public à la place de private ?

Cela permet de rendre le code plus clair et d'éviter les erreurs. On laisse en général les champs en private pour être sûr qu'ils ne seront pas modifiés n'importe comment. Ensuite, on peut néanmoins vouloir accéder à ces champs. Si tel est le cas, on utilise les propriétés pour contrôler l'accès aux champs. Dans certains langages, on parle d'**accesseurs**. En C#, les accesseurs sont get et set. Ils sont utilisés au sein d'une propriété.
Je reprends l'exemple ci-dessus :

```
private ushort m_age;
public ushort Age
{
   get { return m_age; }
   set { m_age = value; }
}
```

Comme je vous l'ai dit plus haut, get gère l'accès en **lecture** alors que set gère l'accès en **écriture**. Si vous ne voulez pas autoriser l'accès en écriture, il suffit de supprimer le set :

```
private ushort m_age;
public ushort Age
{
   get { return m_age; }
}
```

Le 1er morceau de code peut se simplifier en utilisant les accesseurs auto-implémentés :

```
public ushort Age { get; set; }
```

Dans ce cas, vous n'avez plus besoin de m_age. Le compilateur comprend que vous autorisez l'accès en lecture et en écriture.
Si vous utilisez les accesseurs auto-implémentés, vous devez gérer les deux accesseurs.

```
// Erreur du compilateur :
// 'ConsoleApplication1.Person.Age.get' must declare a body because it is not marked abstract or extern.
// Automatically implemented properties must define both get and set accessors.
public ushort Age { get; }
```

Comment faire si je ne veux pas autoriser l'accès en écriture dans le code simplifié ?

Il suffit de rajouter private devant set pour indiquer que Age n'aura le droit d'être modifié qu'à l'intérieur de la classe :

```
public ushort Age { get; private set; }
```

Dans l'écriture simplifiée, **il n'est plus question de m_age**. Considérons le code suivant :

```
private ushort m_age;
public ushort Age { get; private set; }
```

Si je modifie Age, cela ne va pas affecter m_age ! En effet, dans l'écriture simplifiée on ne fait pas de lien entre m_age et Age : ils vivent leur vie chacun de leur côté.
Vous pouvez utiliser les deux écritures dans votre code : soit l'écriture complète, soit l'écriture simplifiée. Pour l'instant je vais rester sur la première que vous compreniez bien ce qui se passe. Et ne mélangez pas les deux !

Par contre, il faut respecter la convention voulant qu'avec l'écriture simplifiée, on écrirait this.Age dans la classe Person au lieu de m_age. Le this est facultatif dans les deux cas mais on l'utilise quand même dans l'écriture simplifiée car il permet de voir que Age dépend de l'instance avec laquelle on travaille.

1.4.3. La comparaison d'objets

Vous pouvez comparer des objets de diverses manières suivant ce que vous voulez vérifier.

Peut-on utiliser l'opérateur == ?

Oui et non ; en fait cela dépend de ce que vous voulez savoir. Considérons l'exemple suivant :

```
Person p1 = new Person("Toto", 10);
Person p2 = new Person("Toto", 10);
Person p3 = p1;
```

Dans cet exemple, p1 n'est pas égal à p2, mais p1 est égal à p3 :

```
Console.WriteLine(p1 == p2 ? "p1 est égal à p2." : "p1 n'est pas égal à p2.");
Console.WriteLine(p1 == p3 ? "p1 est égal à p3." : "p1 n'est pas égal à p3.");
```

Résultat :

```
p1 n'est pas égal à p2.
p1 est égal à p3.
```

En effet, les références p1 et p2 désignent chacune une instance différente de la classe Person. Ces deux instances sont identiques du point de vue de leurs valeurs, mais elles sont bien distinctes. Ainsi, si je modifie la deuxième instance, cela ne va pas affecter la première :

```
Console.WriteLine(p1.Age);
Console.WriteLine(p2.Age);
p2.Age = 5;
Console.WriteLine(p1.Age);

Console.WriteLine(p2.Age);
```

Ce code affichera :

```
10
10
10
5
```

Par contre, les références p1 et p3 sont identiques et désignent donc la même instance. Si je modifie p3, cela affecte donc p1 :

```
Console.WriteLine(p1.Age);
Console.WriteLine(p3.Age);
p3.Age = 42;
Console.WriteLine(p1.Age);
Console.WriteLine(p3.Age);
```

Ce code affichera :

```
10
10
42
42
```

1.5.Des méthodes importantes

1.5.1. Equals

Pour reprendre ce qui vient d'être dit, plutôt que d'utiliser l'opérateur ==, vous pouvez utiliser la méthode Equals (que possède tout objet). La seule différence est que == ne compare que des

objets de même type, alors que Equals permet de comparer un objet d'un certain type avec un autre objet d'un autre type.
Quel est l'intérêt de Equals, sachant que de toute façon si deux variables ne sont pas du même type, elles ne peuvent pas être égales ?

Cela sert si, à l'écriture du code, vous ne connaissez pas le type des variables concernées.

1.5.2. ToString

Tout objet possède aussi cette méthode. Par défaut, elle renvoie le type de l'objet en question :

```
Console.WriteLine(p1.ToString());
```

Résultat :

```
ConsoleApplication1.Person
```

En fait, pour cette ligne de code, il est inutile d'utiliser ToString car de toute façon la méthode WriteLine fait appel à la méthode ToString lorsqu'elle reçoit un objet. J'aurais donc pu écrire Console.WriteLine(p1);.
La méthode ToString peut être modifiée pour retourner un autre résultat. Par exemple ici on pourrait vouloir renvoyer le nom de la personne en question. Cela se fait avec le modificateur override, que nous étudierons plus tard.

1.6.Le modificateur "static"

Le modificateur static se place avant le nom de la classe ou du membre qu'il affecte. Quand vous créez une classe avec des membres, vous faites un moule à objets : à partir de cette classe on fabrique ce qui s'appelle des **instances de la classe**.
Par exemple si je crée une classe qui contient une chaîne de caractères en champ, chaque instance de la classe aura sa propre version de cette chaîne de caractères.
Le mot-clef static sert à faire que ce qui suit ne dépende pas d'une instance, mais dépende de la classe elle-même.
Par habitude, mettez "s_" devant une variable statique. Cela permet d'accroître la lisibilité de votre code.
Voici un exemple :

```
public class Person
{
   private static int s_count;
   public static int Count
   {
      get { return s_count; }
   }

   public Person()
   {
      s_count++;
   }
}
```

L'ordre des modificateurs a de l'importance. static private poserait problème.
J'espère que vous reconnaissez une classe assez simple, avec un léger changement : la présence d'un champ statique s_count que l'on incrémente à chaque appel du constructeur. En

fait j'ai fait un compteur qui indique le nombre d'objets créés. Ce compteur ne dépend pas d'une instance en particulier, mais de la classe entière.
Pour accéder à des champs statiques, il faut écrire : nomDeLaClasse.nomDuChamp. Ici, il faut écrire Person.Count pour accéder à s_count qui est mis en private.
Étant donné que ce champ ne dépend pas d'une instance en particulier, si je crée une instance de la classe Person nommée toto, toto.Count n'existera pas.
On peut aussi utiliser ce modificateur pour une méthode :

```
public class Person
{
    private static int s_count;
    public static int Count
    {
        get { return s_count; }
    }

    public Person()
    {
        s_count++;
    }

    public static void DisplayCount()
    {
        Console.WriteLine(s_count + " objet(s) a(ont) été créé(s).");
    }
}
```

Dans ce cas, vous pouvez ensuite écrire Person.DisplayCount();.
Améliorons cette méthode avec des if :

```
public static void DisplayCount()
{
    if (s_count == 0)
    {
        Console.WriteLine("Aucun objet n'a été créé !");
    }
    else if (s_count == 1)
    {
        Console.WriteLine("Un seul objet a été créé.");
    }
    else if (s_count > 1)
    {
        Console.WriteLine(s_count + " objets ont été créés.");
    }
    else
    {
        Console.WriteLine("Erreur : s_count est négatif ! Il vaut : " + s_count);
    }
}
```

Au cas où tous ces Console.WriteLine ne vous auraient pas fait tilt, eh oui, il s'agit bien de l'appel à la méthode statique WriteLine de la classe Console.

Vous pouvez créer une classe entièrement statique, dont le but n'est alors pas de servir de moule à instances. C'est d'ailleurs le cas de la classe Console : on ne veut pas créer d'objet de type Console, mais faire des choses avec celle qui est ouverte. C'est aussi le cas de la classe Convert qui sert à faire des conversions et que nous verrons sous peu. Pour faire cela il faut mettre static devant le nom de la classe, et du coup devant chacun de ses membres :

```
public static class MyStaticClass
{
    public static int s_myStaticInt;

    public static string TellSomething()
    {
        return "This is all static!";
    }
}
```