

CHAPITRE 3

CLASSE ET ENCAPSULATION

3.1 INTRODUCTION AU CONCEPT DE CLASSE

Le but général de la programmation orienté objet est de mettre les classes en relation d'appartenance (a-un) ou d'héritage (est-un) entre elles.

Dans ce chapitre, nous allons explorer ensemble le concept de classe, des constructeurs, des classes et membres statiques, de propriétés, de classes partielles et de l'encapsulation afin de vous préparer aux notions d'héritage, de polymorphisme, de classes abstraites et d'interface dans le chapitre qui suivra.

Vous êtes déjà familiés de par votre cours INF1002 au concept objet, mais évidemment ici non seulement nous le réviserons mais nous verrons les différentes particularités de C#. De bons exemples sont la notion de propriétés (et de propriétés automatiques) et l'initialisation d'objets sans passer par un constructeur.

Bien évidemment le type central au framework .NET est la classe. On la définit comme un type personnalisé en tant que structure de donnée composée de **champs de données** (qu'on appelle aussi des variables membres ou des variables d'instances) et des **membres** (comme des méthodes, des événements et des propriétés) qui opèrent sur ces données (qui ensemble constituent les fonctionnalités, aussi appelées les services, que fournit la classe en question.

Collectivement, l'ensemble des champs de données représentent l'état d'une instance d'une classe, qu'on appelle un **objet**.

Vous avez certainement vu l'analogie avec le plan d'une maison qui contient tous ses paramètres (la classe et ses variables) et la maison construit selon certaines variations permises par les paramètres (l'instanciation du plan de la maison qui forme un objet de maison et ses caractéristiques (son état)). Ce n'est certes pas différent en C#.

À la base, la classe se définit à la façon d'une structure. On retrouvera ainsi le nom de la classe, puis à un niveau inférieur les membres de la classe, dont les champs de données – dont les valeurs sont généralement propres à chaque instance de la classe – et les autres catégories de membres comme les propriétés et les méthodes.

Nous avons déjà créé à cet effet une classe `Point2` au chapitre précédent, mais redonnons un second exemple pour une classe que nous appellerons, disons, `Superhero` :

```
class Superhero
{
    //-- ces champs composent l'état d'un superhéro --
    public string nom;
    public string alias; // vrai nom
    public string superPouvoir;
    public string ligneSignature; // pour provoquer l'ennemi
    public int force;
    public int vitesse;
    public int vitesseActuelle;
    public bool peutVoler;

    //-- ces membres composent les fonctionnalités fournies par un superhero -
    public void AfficherEtat()
    {
        Console.WriteLine("{0}, alias {1}, a le super pouvoir de {2}. Pour
        provoquer ses ennemis, il aime bien lancer la phrase suivante:
        \"{3}\".", nom, alias, superPouvoir, ligneSignature);
        Console.WriteLine("Il a {0} de force et {1} de vitesse (sa vitesse
        actuelle est de {3}). Il {2} voler.", force, vitesse, peutVoler ?
        "peut" : "ne peut pas", vitesseActuelle);
    }

    public void ChangerVitesse(int delta)
    {
        if (vitesseActuelle + delta < 0)
            vitesseActuelle = 0;
        else if (vitesseActuelle + delta > vitesse)
            vitesseActuelle = vitesse;
        else
            vitesseActuelle += delta;
    }
}
```

Et si on veut l'utiliser, il suffit de déclarer une variable du type `Superhero`, de l'instanciation grâce au mot clé `new`, puis dans notre cas donner les valeurs pour les variables d'instances :

```
static void Main(string[] args)
{
    Superhero superhero = new Superhero();
    superhero.nom = "Super Parici";
    superhero.alias = "James Sergon";
    superhero.vitesse = 10;
    superhero.vitesseActuelle = 0;
    superhero.force = 5;
    superhero.superPouvoir = "Métamorphose de la matière en banane";
}
```

```
superhero.ligneSignature = "Sous nos pelures nous sommes tous des  
bananes qui ne demandent qu'à être épluchées!";  
  
for (int i = 0; i < 4; i++)  
{  
    superhero.ChangerVitesse(2);  
    superhero.AfficherEtat();  
}  
  
Console.ReadLine();  
}
```

Nous appelons à la toute fin la méthode `ChangerVitesse()` en boucle à quelques reprises de façon à démontrer qu'effectivement nous parvenons à altérer l'état de l'objet référé par la variable `superhero`.

En sortie, sans grande surprise, nous avons alors :

```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...  
Super Parici, alias James Sergon, a le super pouvoir de Métamorphose de la matiè  
re en banane. Pour provoquer ses ennemis, il aime bien lancer la phrase suivante  
: "Sous nos pelures nous sommes tous des bananes qui ne demandent qu'à être épel  
uchées!".  
Il a 5 de force et 10 de vitesse (sa vitesse actuelle est de 2). Il ne peut pas  
voler.  
Super Parici, alias James Sergon, a le super pouvoir de Métamorphose de la matiè  
re en banane. Pour provoquer ses ennemis, il aime bien lancer la phrase suivante  
: "Sous nos pelures nous sommes tous des bananes qui ne demandent qu'à être épel  
uchées!".  
Il a 5 de force et 10 de vitesse (sa vitesse actuelle est de 4). Il ne peut pas  
voler.  
Super Parici, alias James Sergon, a le super pouvoir de Métamorphose de la matiè  
re en banane. Pour provoquer ses ennemis, il aime bien lancer la phrase suivante  
: "Sous nos pelures nous sommes tous des bananes qui ne demandent qu'à être épel  
uchées!".  
Il a 5 de force et 10 de vitesse (sa vitesse actuelle est de 6). Il ne peut pas  
voler.  
Super Parici, alias James Sergon, a le super pouvoir de Métamorphose de la matiè  
re en banane. Pour provoquer ses ennemis, il aime bien lancer la phrase suivante  
: "Sous nos pelures nous sommes tous des bananes qui ne demandent qu'à être épel  
uchées!".  
Il a 5 de force et 10 de vitesse (sa vitesse actuelle est de 8). Il ne peut pas  
voler.
```

L'utilisation du mot clé `new` est quelque peu différente que dans une structure.

Lorsque nous avons l'exemple de la structure `Point`, nous avons vu que nous pouvions avoir le code suivant sans problème :

```
Point p;  
p.X = 4;  
p.Y = 9;  
p.Afficher();
```

C'était dû au fait qu'une structure est un type valeur.

Dans le cas d'une classe, qui est un type référence, les objets doivent être instanciés en mémoire en utilisant le mot clé `new`. Cela a pour effet de créer l'espace nécessaire pour conserver l'état de l'objet dans le tas, puis de stocker la référence vers cette plage en mémoire pour la variable de classe.

Ceci génère donc une erreur de compilation :

```
Superhero superhero;  
superhero.nom = "Super Parici";  
(local variable) Superhero superhero  
  
Error:  
Use of unassigned local variable 'superhero'
```

Il faut plutôt faire comme dans le code original, ou retarder l'utilisation du `new` pour créer l'instance puis la référer en l'affectant à la variable :

```
Superhero superhero;  
superhero = new Superhero();  
superhero.nom = "Super Parici";  
(...)
```

Dans ce cas, la première ligne de code réserve sur la pile un espace mémoire suffisant pour stocker une adresse mémoire (la variable vaut `null` à ce moment), jusqu'à ce qu'on lui affecte une référence vers un objet valide en mémoire (deuxième ligne).

Le `new` fait intervenir la notion de constructeur, qui constitue le prochain sujet à traiter dans ce chapitre.

3.2 CONSTRUCTEURS

Nous avons déjà vu en surface avec les structures (et l'exemple de `Point`) l'utilisation de `new`, qui appelle une méthode « spéciale » portant le même nom que le type et dans lequel on peut possiblement avoir des paramètres.

Le rôle des constructeurs consiste à pouvoir initialiser l'état d'un objet nouvellement créé avec des valeurs qui peuvent ou non dépendre de certaines autres valeurs passées en paramètres (du constructeur).

Le but est d'éviter à devoir faire comme dans l'exemple précédent, c'est-à-dire affecter une à une des valeurs initiales pour chacune des variables d'instances pour l'objet (pointée par la variable `superhero` dans notre cas).

Un des autres intérêts des constructeurs va de paire avec le respect de l'encapsulation, que nous explorerons plus loin dans ce chapitre.

Une classe pourra avoir de 0 à n constructeurs définis par le développeur, dont les contraintes syntaxiques correspondent à celles des méthodes, auxquelles s'ajoutent ces contraintes-ci :

- Ils n'ont pas de mot clé pour des valeurs de retour;
- Ils portent tous comme nom le nom exact du type.

3.2.1 Constructeur par défaut

Chaque classe vient avec un constructeur implicite par défaut qui n'a pas à être défini, mais que vous pouvez redéfinir si vous le souhaitez. Sachez que :

- Un constructeur par défaut n'a aucun paramètre;
- Un constructeur par défaut s'assure que tous les champs de données de la classe se font attribués une valeur par défaut appropriée (0 pour les types valeurs, `null` pour les types références) suite à l'allocation en mémoire de l'objet.

Si les valeurs par défaut ne sont pas désirables pour certains ou l'ensemble des variables d'instances de la classe, vous pouvez redéfinir ce constructeur par défaut.

Si par exemple on intègre les affectations initiales de l'exemple précédent mais dans un constructeur par défaut pour la classe `Superhero`, on obtient le code suivant :

```
class Superhero
{
    //-- ces champs composent l'état d'un superhéro --
    public string nom;
    public string alias; // vrai nom
    public string superPouvoir;
    public string ligneSignature; // pour provoquer l'ennemi
    public int force;
    public int vitesse;
    public int vitesseActuelle;
    public bool peutVoler;

    public Superhero()
```

```
{
    nom = "Super Parici";
    alias = "James Sergon";
    vitesse = 10;
    vitesseActuelle = 0;
    force = 5;
    superPouvoir = "Métamorphose de la matière en banane";
    ligneSignature = "Sous nos pelures nous sommes tous des bananes qui ne
    demandent qu'à être épluchées!";
}

//-- ces membres composent les fonctionnalités fournies par un superhero -
-
public void AfficherEtat()
{
    (...)
}

public void ChangerVitesse(int delta)
{
    (...)
}
}
```

Cela allège évidemment le code si on avait à créer plusieurs objets de type Superhero :

```
Superhero superhero, superhero2;
superhero = new Superhero();
superhero2 = new Superhero();
```

3.2.2 Constructeur personnalisé

Très souvent, on voudra faire en sorte que les valeurs par défaut pour nos champs de données soient plus flexibles et dépendent du contexte. On devra alors avoir recours à des constructeurs personnalisés qui contiennent des paramètres qui expriment ce contexte d'initialisation de l'objet.

Dans cette optique, nous pourrions très bien doter notre classe Superhero avec les constructeurs suivant :

```
public Superhero()
{
    nom = "Super Parici";
    alias = "James Sergon";
    vitesse = 10;
    vitesseActuelle = 0;
    force = 5;
    superPouvoir = "Métamorphose de la matière en banane";
    ligneSignature = "Sous nos pelures nous sommes tous des bananes qui ne
    demandent qu'à être épluchées!";
}
```

```
}

public Superhero(string n, string a, int vitesseMax, int vitesseDepart)
{
    nom = n;
    alias = a;
    vitesse = vitesseMax;
    vitesseActuelle = vitesseDepart;
    force = 5; // cette valeur est celle par "défaut" lorsqu'on utilise ce
               // constructeur
    //ligneSignature, canFly et superPouvoir non définis: vont recevoir
    null ou false comme valeurs par défaut
}

public Superhero(string n, string a, int vitesseMax, int vitesseDepart,
int str, string pouvoir, string ligneFetiché, bool canFly)
{
    nom = n;
    alias = a;
    vitesse = vitesseMax;
    vitesseActuelle = vitesseDepart;
    force = str;
    superPouvoir = pouvoir;
    ligneSignature = ligneFetiché;
    peutVoler = canFly;
}
```

Comme vous pouvez le constater, la signature de chaque constructeur est unique, et nous sommes en présence de surcharge (de méthodes).

La flexibilité vient donc non seulement des valeurs des paramètres (le contexte) mais aussi du choix du constructeur.

En sortie, avec le code suivant dans le `Main()` :

```
static void Main(string[] args)
{
    Superhero[] superheros = new Superhero[3];

    superheros[0] = new Superhero();
    superheros[1] = new Superhero("Super Sacabou", "Jack Brown", 8, 2);
    superheros[2] = new Superhero("Super Sen tai", "Betty Rave", 3, 1, 4,
    "Tempête de céréales biologiques", "Si ça goûte vert, c'est bon pour
    ton corps", false);

    foreach (Superhero sh in superheros)
        sh.AfficherEtat();

    Console.ReadLine();
}
```

Nous aurions donc ceci en console :

```

file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
Super Parici, alias James Sergon, a le super pouvoir de Métamorphose de la matiè
re en banane. Pour provoquer ses ennemis, il aime bien lancer la phrase suivante
: "Sous nos pelures nous sommes tous des bananes qui ne demandent qu'à être épel
uchées!".
Il a 5 de force et 10 de vitesse <sa vitesse actuelle est de 0>. Il ne peut pas
voler.
Super Sacabou, alias Jack Brown, a le super pouvoir de . Pour provoquer ses enne
mis, il aime bien lancer la phrase suivante: "".
Il a 5 de force et 8 de vitesse <sa vitesse actuelle est de 2>. Il ne peut pas
voler.
Super Sen tai, alias Betty Rave, a le super pouvoir de Tempête de céréales biolo
giques. Pour provoquer ses ennemis, il aime bien lancer la phrase suivante: "Si
ça goûte vert, c'est bon pour ton corps".
Il a 4 de force et 3 de vitesse <sa vitesse actuelle est de 1>. Il ne peut pas
voler.

```

3.2.3 Constructeur par défaut (bis)

Supprimons (ou plutôt entre commentaires) dans notre exemple la redéfinition du constructeur par défaut (celui sans paramètres). Si vous vous servez de l'Intelli-Sense ou que vous tentez de compiler une application qui tente de faire appel à un constructeur par défaut pour initialiser un objet de ce type, vous aurez l'erreur suivante dans Visual Studio :

```

superheros[0] = new Superhero();
superheros[1] = 'Chapitre2.Superhero' does not contain a constructor that takes 0 arguments
superheros[2] =

```

Le constructeur par défaut n'est pas accessible!

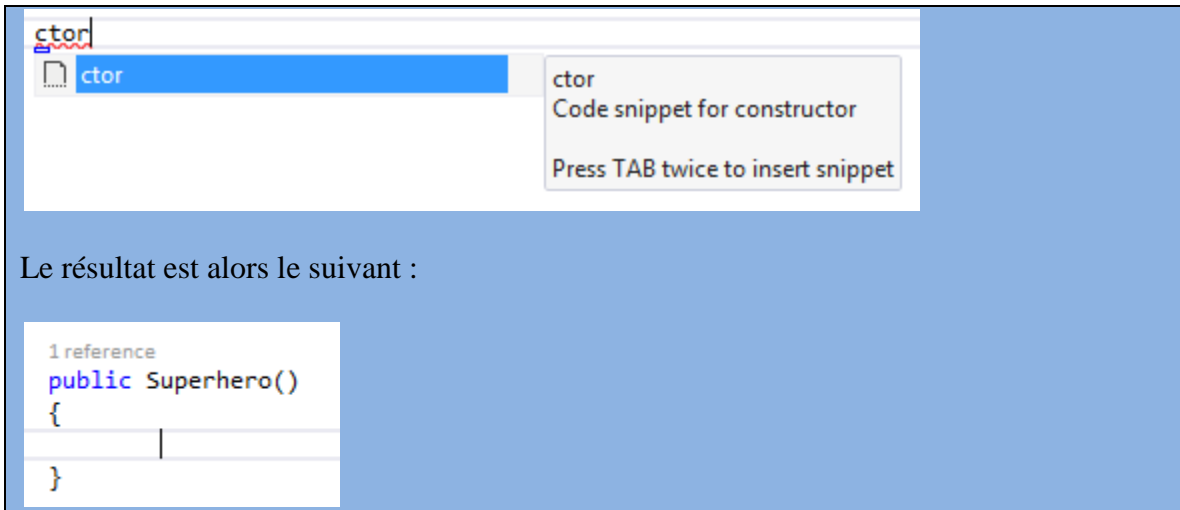
En fait, **à partir du moment où vous avez un ou plusieurs constructeurs personnalisés** (sans avoir redéfini le constructeur par défaut), **le compilateur considère que vous avez pris en charge l'initialisation de l'objet** vous-mêmes.

Ainsi, si vous souhaitez permettre la création d'une instance d'une classe via le constructeur par défaut, il est obligatoire de redéfinir celui-ci.

Note :

Vous pouvez utiliser un *snippet* (c'est-à-dire un code raccourci) avec Visual Studio qui vous permet de définir automatiquement pour vous le squelette d'un constructeur.

Pour cela, il suffit de taper « `ctor` » et appuyer deux fois sur « *Tab* ».



3.2.4 Paramètres optionnels avec les constructeurs

Rien ne vous empêche non plus de rendre des paramètres de vos constructeurs optionnels.

Une façon de s'en servir pourrait être de donner des valeurs par défaut aux différents paramètres d'un constructeur « principal », de façon à ne pas avoir besoin de redéfinir un constructeur par défaut « à part » et même certains autres constructeurs supplémentaires.

Par exemple, nous pouvons modifier notre code pour `Superhero` de façon à regrouper tous les constructeurs sous un seul constructeur sans effet apparent à l'exécution (il faut aussi mettre tous les paramètres optionnels à la fin, rappelez-vous) :

```
class Superhero
{
    (...champs...)

    public Superhero(string n, string a, string pouvoir, string ligneFetiché,
        int vitesseMax = 0, int vitesseDépart = 10, int str = 5, bool canFly =
        true)
    {
        nom = n;
        alias = a;
        vitesse = vitesseMax;
        vitesseActuelle = vitesseDépart;
        force = str;
        superPouvoir = pouvoir;
        ligneSignature = ligneFetiché;
        peutVoler = canFly;
    }

    (...autres méthodes...)
```

```
}
```

On peut donc du côté appelant créer des instances équivalentes de cette façon :

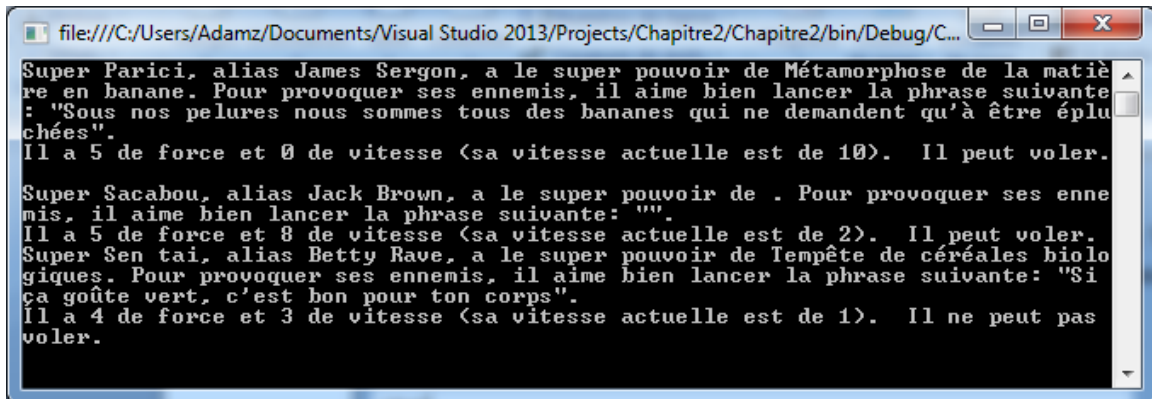
```
static void Main(string[] args)
{
    Superhero[] superheroes = new Superhero[3];

    superheroes[0] = new Superhero("Super Parici", "James Sergon",
    "Métamorphose de la matière en banane", "Sous nos pelures nous sommes
    tous des bananes qui ne demandent qu'à être épluchées");
    superheroes[1] = new Superhero("Super Sacabou", "Jack Brown", null,
    null, 8, 2);
    superheroes[2] = new Superhero("Super Sen tai", "Betty Rave", "Tempête
    de céréales biologiques", "Si ça goûte vert, c'est bon pour ton
    corps", 3, 1, 4, false);

    foreach (Superhero sh in superheroes)
        sh.AfficherEtat();

    Console.ReadLine();
}
```

Ce qui mène à la même sortie :



3.3 THIS

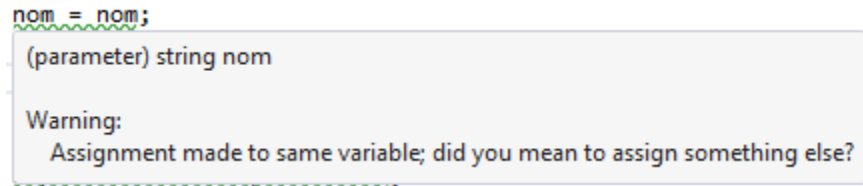
Comme en Java, le mot clé `this` en C# sert à donner explicitement accès à l'instance courante de la classe.

Le but général est de résoudre les cas d'ambiguïté où, par exemple, une variable locale ou un paramètre porterait le même nom que la variable d'instance.

Par exemple, ceci causerait des cas d'ambiguïtés :

```
public Superhero(string nom, string alias, string superPouvoir, string
ligneSignature, int vitesse = 0, int vitesseDepart = 10, int force = 5,
bool peutVoler = true)
{
    nom = nom;
    alias = alias;
    vitesse = vitesse;
    vitesseActuelle = vitesseDepart;
    force = force;
    superPouvoir = superPouvoir;
    ligneSignature = ligneSignature;
    peutVoler = peutVoler;
}
```

L'avertissement que vous sert Visual Studio ressemble alors à ceci :



3.3.1 Syntaxe générale

Plutôt, on voudra spécifier `this` devant les variables d'instance pour résoudre le tout :

```
public Superhero(string nom, string alias, string superPouvoir, string
ligneSignature, int vitesse = 0, int vitesseDepart = 10, int force = 5,
bool peutVoler = true)
{
    this.nom = nom;
    this.alias = alias;
    this.vitesse = vitesse;
    this.vitesseActuelle = vitesseDepart;
    this.force = force;
    this.superPouvoir = superPouvoir;
    this.ligneSignature = ligneSignature;
    this.peutVoler = peutVoler;
}
```

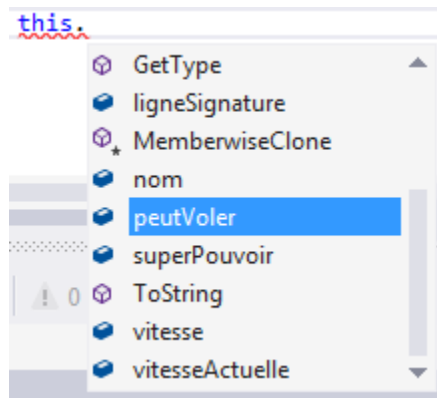
On peut s'en servir n'importe où dans notre type, qu'il y ait ambiguïté ou non, comme par exemple :

```
public void ChangerVitesse(int delta)
{
    if (this.vitesseActuelle + delta < 0)
        this.vitesseActuelle = 0;
    else if (this.vitesseActuelle + delta > this.vitesse)
        this.vitesseActuelle = vitesse;
}
```

```
else
    this.vitesseActuelle += delta;
}
```

Bien que cela soit facultatif, **l'intérêt de l'utiliser dans tous les cas** (c'est-à-dire même les cas non ambigus) où on fait appel à une variable d'instance est double :

- D'une part, en tapant `this.`, vous aurez **accès à l'Intelli-Sense** sur tous vos membres de la classe, ce qui est pratique lorsqu'on programme :



- D'un point de vue de **lisibilité**, celui qui lit le code sait à tout moment si la variable concernée en est une d'instance; cela permet donc de distinguer à la volée les données des objets manipulées par les méthodes ou utilisées dans des traitements.

Sachez enfin que par défaut si vous utilisez le nom d'une variable sans le `this`, le compilateur comprendra que vous faites référence au paramètre de méthode ou à la variable locale si elle porte le même nom qu'une variable d'instance.

3.3.2 Enchaînement de constructeurs

Bien que la possibilité d'avoir des paramètres optionnels rend la seconde utilisation de `this` obsolète dans la plupart des cas, sachez que vous pouvez aussi utiliser `this()` pour appeler des constructeurs particuliers. Il suffit alors d'ajouter les paramètres du constructeur en particulier entre les parenthèses suite à l'entête du constructeur à relayer.

Le but est, comme pour les paramètres optionnels, d'éviter de la redondance dans votre code. Cette technique est par ailleurs ce à quoi on fait référence lorsqu'on parle « d'enchaînement de constructeurs ».

Imaginons les deux constructeurs suivants (sans utiliser des paramètres optionnels) :

```
public Superhero(string nom, string alias, int vitesse, int
vitesseActuelle)
{
    this.nom = nom;
    this.alias = alias;
    this.vitesse = vitesse;

    if (vitesseActuelle <= this.vitesse)
        this.vitesseActuelle = vitesseActuelle;
    else
        this.vitesseActuelle = this.vitesse;

    this.force = 5;
}

public Superhero(string nom, string alias, int vitesse, int
vitesseActuelle, int force, string superPouvoir, string ligneSignature,
bool peutVoler)
{
    this.nom = nom;
    this.alias = alias;
    this.vitesse = vitesse;

    if (vitesseActuelle <= this.vitesse)
        this.vitesseActuelle = vitesseActuelle;
    else
        this.vitesseActuelle = this.vitesse;

    this.force = force;
    this.superPouvoir = superPouvoir;
    this.ligneSignature = ligneSignature;
    this.peutVoler = peutVoler;
}
```

On s'assure que la vitesse de départ ne dépasse pas la vitesse maximale déterminée. On peut observer que la vérification est redondante dans les deux cas. À la limite, on peut considérer les affectations directes communes comme également des redondances. Si le code a à être modifié, il devra donc l'être à plusieurs endroits, ce qui n'est pas idéal.

Une première approche pourrait consister à avoir une méthode qui sert à gérer cette règle spécifique concernant la vitesse actuelle :

```
public Superhero(string nom, string alias, int vitesse, int
vitesseActuelle)
{
    this.nom = nom;
    this.alias = alias;
    this.vitesse = vitesse;

    ValiderVitesseActuelle(vitesseActuelle);
}
```

```
        this.force = 5;
    }

    public Superhero(string nom, string alias, int vitesse, int
vitesseActuelle, int force, string superPouvoir, string ligneSignature,
bool peutVoler)
    {
        this.nom = nom;
        this.alias = alias;
        this.vitesse = vitesse;

        ValiderVitesseActuelle(vitesseActuelle);

        this.force = force;
        this.superPouvoir = superPouvoir;
        this.ligneSignature = ligneSignature;
        this.peutVoler = peutVoler;
    }

    public void ValiderVitesseActuelle(int vitesseActuelle)
    {
        if (vitesseActuelle <= this.vitesse)
            this.vitesseActuelle = vitesseActuelle;
        else
            this.vitesseActuelle = this.vitesse;
    }
}
```

Bien que dans ce cas s'il y a un changement de règle de validation la mise à jour n'a qu'à se faire dans la méthode `ValiderVitesseActuelle()` et non dans chacun des constructeurs, il reste toutefois une certaine redondance.

Elle peut être éliminée en utilisant la technique d'enchaînement de constructeurs, où **on a un ou plusieurs constructeurs généralement « vides d'instructions » qui ne font qu'appeler un constructeur « maître » qui fait tout le « travail »**. Dans certains cas, on pourra également un enchaînement multiple qui passe par plusieurs constructeurs intermédiaire jusqu'au constructeur maître, mais ce n'est pas toujours utile de procéder ainsi.

Dans l'exemple de code qui suit, on peut observer les deux situations (remarquez bien la syntaxe) :

```
public Superhero(string nom)
    : this(nom, "Inconnu") { }

public Superhero(string nom, string alias)
    : this(nom, alias, 10, 0, 5, "Inconnu", "Inconnue", false) { }

public Superhero(string nom, string alias, int vitesse, int
vitesseActuelle)
```

```
        : this(nom, alias, vitesse, vitesseActuelle, 5, "Inconnu", "Inconnue",
false) { }

public Superhero(string nom, string alias, int vitesse, int
vitesseActuelle, int force, string superPouvoir, string ligneSignature,
bool peutVoler)
{
    this.nom = nom;
    this.alias = alias;
    this.vitesse = vitesse;

    if (vitesseActuelle <= this.vitesse)
        this.vitesseActuelle = vitesseActuelle;
    else
        this.vitesseActuelle = this.vitesse;

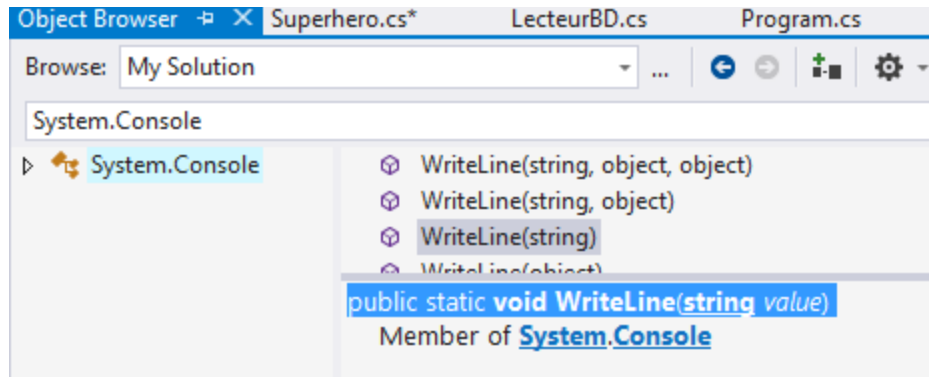
    this.force = force;
    this.superPouvoir = superPouvoir;
    this.ligneSignature = ligneSignature;
    this.peutVoler = peutVoler;
}
```

3.4 STATIC

Comme vous le savez sans doute, toute classe peut définir un nombre virtuellement non limité de membres statiques (des champs, des méthodes (incluant des constructeurs) en utilisant le mot clé `static`. Les classes peuvent elles-mêmes être statiques.

Lorsque vous invoquez de tels membres, vous vous rappelez probablement que vous devez les appeler directement du niveau de la classe, et non en utilisant la variable qui fait référence à l'instance d'une classe de ce type.

L'exemple vu le plus souvent jusqu'ici concerne certainement l'utilisation de la console, pour laquelle on appelle directement `WriteLine()` ou `ReadLine()`, qui sont des méthodes statiques définies dans la classe (elle-même statique) `System.Console`, comme en fait foi aussi l'afficheur d'objets :



Bien que toute classe soit libre de définir des membres statiques, ils sont le plus souvent retrouvés à l'intérieur de classes utilitaires, souvent statiques.

On entend par **classe utilitaire** toute classe qui ne maintient aucun état lié à une instance quelconque et qui expose toutes ses fonctionnalités en tant que membres statiques.

D'autres exemples de classes utilitaires sont `System.Math` et `System.Environment`.

3.4.1 Champs statiques

Considérez les variables d'instances comme étant des « variables non statiques ». Il s'agit là du type le plus commun de données, qui sont attachées à l'état d'une instance d'une classe en question.

À l'inverse, lorsqu'une donnée/un champ est défini comme étant statique, la donnée en mémoire est partagée par toutes les instances du type en question. La donnée appartient donc en quelque sorte à la classe et non à un objet en particulier de la classe.

Une des raisons de vouloir utiliser une variable de classe (statique) peut donc consister à garder en mémoire des données qui concernent toutes instances courantes ou futures de la classe, généralement parce qu'elle sera utilisée dans les algorithmes et pourra avoir une influence sur l'état des instances ou sur leurs services rendus.

Imaginons une application qui gère des comptes bancaires. Nous pourrions vouloir garder en mémoire pour chacun des comptes la balance courante du compte, mais aussi de façon générale le taux de ristourne. Nous pourrions avoir une classe telle que :

```
class CompteBancaire
{
```



```
public static double tauxRistourne = 0.03; // sur un an; doit être divisé
par 12 à chaque mois avant d'être appliqué
public double balance;

public CompteBancaire()
    : this(1) { } // un compte est toujours ouvert avec un minimum de 1$
public CompteBancaire(double balance)
{
    this.balance = balance;
}

public void AppliquerRistourne()
{
    this.balance *= (1 + tauxRistourne / 12);
}

}
```

Vous auriez pu aussi écrire :

```
this.balance *= (1 + CompteBancaire.tauxRistourne / 12);
```

Puis, du côté du Main (), on pourrait imaginer quelque chose tel que ceci :

```
static void Main(string[] args)
{
    CompteBancaire compteA = new CompteBancaire();
    CompteBancaire compteB = new CompteBancaire(1000);
    CompteBancaire compteC = new CompteBancaire(394.39);

    Console.WriteLine("Compte A: {0:c}", compteA.balance);
    Console.WriteLine("Compte B: {0:c}", compteB.balance);
    Console.WriteLine("Compte C: {0:c}", compteC.balance);

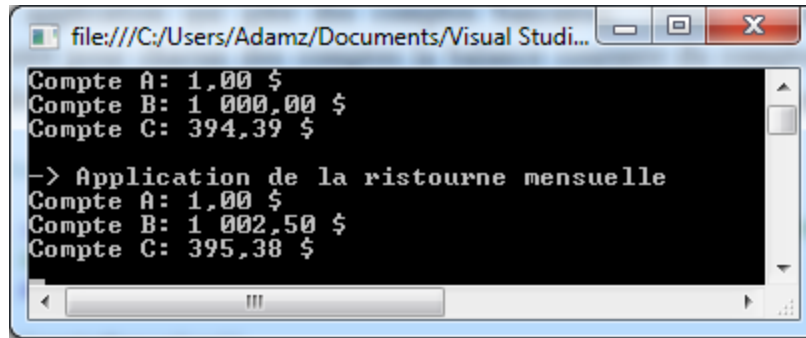
    Console.WriteLine("\n-> Application de la ristourne mensuelle");

    compteA.AppliquerRistourne();
    compteB.AppliquerRistourne();
    compteC.AppliquerRistourne();

    Console.WriteLine("Compte A: {0:c}", compteA.balance);
    Console.WriteLine("Compte B: {0:c}", compteB.balance);
    Console.WriteLine("Compte C: {0:c}", compteC.balance);

    Console.ReadLine();
}
```

On aura en sortie :



```
file:///C:/Users/Adamz/Documents/Visual Studi...
Compte A: 1,00 $
Compte B: 1 000,00 $
Compte C: 394,39 $

-> Application de la ristourne mensuelle
Compte A: 1,00 $
Compte B: 1 002,50 $
Compte C: 395,38 $
```

3.4.2 Méthodes statiques

Maintenant, pour changer la valeur d'une variable statique (en considérant une encapsulation), on voudra généralement procéder en utilisant une méthode statique.

Ajoutons deux méthodes statiques qui permettent d'incrémenter ou décrémente par intervalle de 0.25 point le taux de ristournes :

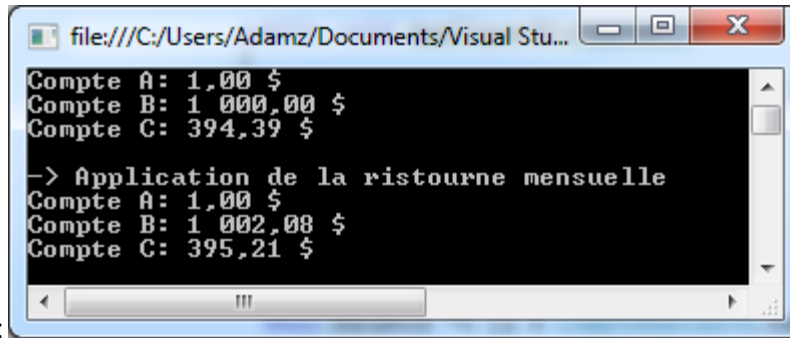
```
public static void IncrementerRistourne()
{
    tauxRistourne += 0.0025;
}

public static void DecrementerRistourne()
{
    if (tauxRistourne > 0 )
        tauxRistourne -= 0.0025;
```

Il suffit alors d'appeler les méthodes statiques au besoin pour manipuler les données statiques :

```
compteA.AppliquerRistourne();
CompteBancaire.DecrementerRistourne();
CompteBancaire.DecrementerRistourne();
compteB.AppliquerRistourne();
compteC.AppliquerRistourne();
```

On aura alors des montants quelque peu différents en sortie :



```

file:///C:/Users/Adamz/Documents/Visual Stu...
Compte A: 1,00 $
Compte B: 1 000,00 $
Compte C: 394,39 $

-> Application de la ristourne mensuelle
Compte A: 1,00 $
Compte B: 1 002,08 $
Compte C: 395,21 $
  
```

Notez que vous ne pouvez évidemment pas accéder à l'intérieur de méthodes statiques à des variables autres que locales ou statiques (donc pas d'accès aux variables d'instance). Ceci est donc illégal :

```

// references
public static void IncrementerRistourne()
{
    tauxRistourne += 0.0025;
    this.balance;
}
double CompteBancaire.balance
2 refe
publ
{
    Error:
    Only assignment, call, increment, decrement, await, and new object expressions can be used as a statement
    if (tauxRistourne > 0)
  
```

3.4.3 Constructeurs statiques

Vous pouvez définir un constructeur statique pour une classe. Celui-ci a pour **rôle d'initialiser les valeurs des variables statiques de la classe de façon sécuritaire**.

Par exemple, considérons la mise à jour suivante pour notre compte bancaire :

```

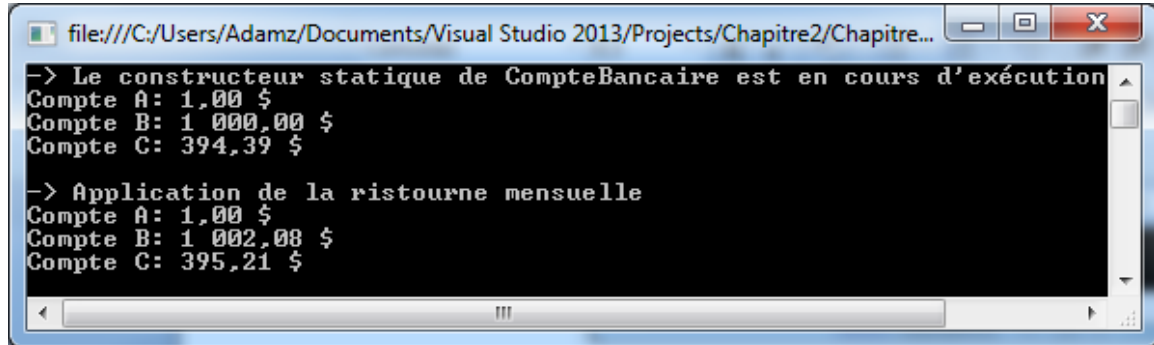
class CompteBancaire
{
    public static double tauxRistourne;
    public double balance;

    static CompteBancaire()
    {
        Console.WriteLine("-> Le constructeur statique de CompteBancaire est
en cours d'exécution");
        tauxRistourne = 0.03;
    }

    public CompteBancaire()
        : this(1) { } // un compte est toujours ouvert avec un minimum de 1$

    (...)
}
  
```

À l'exécution, vous devriez voir ceci :



```
-> Le constructeur statique de CompteBancaire est en cours d'exécution
Compte A: 1,00 $
Compte B: 1 000,00 $
Compte C: 394,39 $

-> Application de la ristourne mensuelle
Compte A: 1,00 $
Compte B: 1 002,08 $
Compte C: 395,21 $
```

Voyez que le constructeur statique est appelé sans qu'on ait à explicitement le demander.

Retenez les éléments suivants concernant les constructeurs statiques :

- **Un seul constructeur statique** peut être défini par classe;
- Un constructeur statique **ne peut pas prendre de paramètres en entrée**;
- Un constructeur statique **ne peut avoir aucun modificateur d'accès** (`public`, `private`, ...);
- Un constructeur statique **s'exécute une seule fois**, peu importe le nombre d'objets de la classe créés, qui est invoqué :
 - **Soit avant de créer une première instance de la classe** (avant l'appel du constructeur pour l'instance);
 - **Soit avant le premier appel à un membre statique** de la classe.

3.4.4 Classes statiques

Enfin, il est possible d'appliquer le mot clé `static` directement au niveau de la classe. Lorsqu'une classe a été définie comme étant statique :

- Aucune instance ne peut être créée avec le mot clé `new`;
- Cette classe statique ne peut alors contenir que des membres de nature également statique.

Nous avons déjà vu des exemples de cela, alors nous ne ferons pas d'exemples plus approfondies. Consultez `System.Math` ou `System.Environment` pour constater cela.

3.5 INTRODUCTION AUX PRINCIPES FONDAMENTAUX DE LA POO

Tous les langages orientés objet sont conçus autour de 3 principes fondamentaux :

- **L'encapsulation** : comment le langage cache-t-il les détails d'implémentation internes d'un objet et préserve l'intégrité de ses données?
- **L'héritage** : comment le langage promouvoit-il la réutilisation de code?
- **Le polymorphisme** : comment le langage permet-il de traiter des objets avec des héritages conjoints d'être traités de façon unifié?

Voyons immédiatement un aperçu rapide de ces trois rôles. Nous les approfondirons en profondeur un à la fois d'ici la fin du prochain chapitre.

3.5.1 Encapsulation

L'encapsulation concerne le fait de cacher les détails d'implémentation d'un objet à l'utilisateur, selon un principe de **boîte noire** :

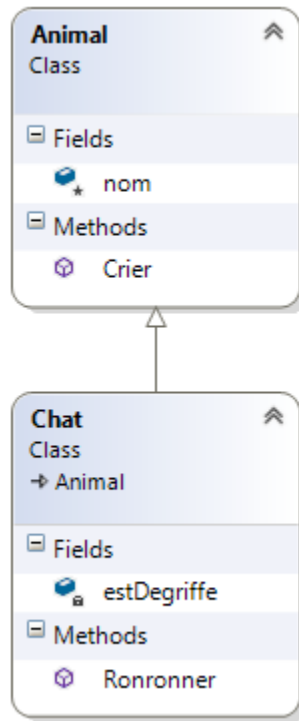
- Le principe de la boîte noire (en opposition avec la boîte de verre, qui est transparente) consiste à **limiter la vision/la connaissance externe sur les services que peut rendre la boîte, ce qui peut rentrer dans la boîte et ce qui peut en ressortir**. On ne s'intéresse pas à ce qui se passe à l'intérieur, c'est-à-dire les détails algorithmes sur la façon dont les services sont rendus.
- Le but est également **de protéger les données/l'état des objets (ou les données des classes) de toute alteration directe** :
 - On n'utilisera pas `public` devant la définition des variables d'instance (c'est une violation du principe d'encapsulation), mais plutôt `private` ou `protected` (nous y reviendrons à la section 3.6);
 - **Pour changer ou accéder aux valeurs de ces variables d'instances**, on passera par des méthodes publiques (appelées des accesseurs et mutateurs) ou, plus communément en C#, par des **propriétés .NET**.

3.5.2 Héritage

Cette caractéristique des langages orientés objet décrit la capacité d'une classe à pouvoir être définie en se basant sur une classe pré-existante, dans le but de l'étendre, c'est-à-dire d'y ajouter de nouvelles données et/ou fonctionnalités. On parlera de « dérivation ». En d'autres termes, il s'agit d'en faire une classe davantage spécialisée.

On aura donc une **relation entre une classe de base** (aussi appelée « classe parent », « classe mère » ou « superclasse ») **et une classe dérivée** (aussi appelée « classe enfant » ou « sous-classe ») **de type « est-un »**.

Par exemple, on peut imaginer la chaîne d'héritage suivante :



Dans ce contexte, un chat aura aussi un nom et pourra crier, mais on aura également un état à savoir s'il est dégriffé ou non et il aura aussi l'habileté de ronronner.

3.5.3 Polymorphisme

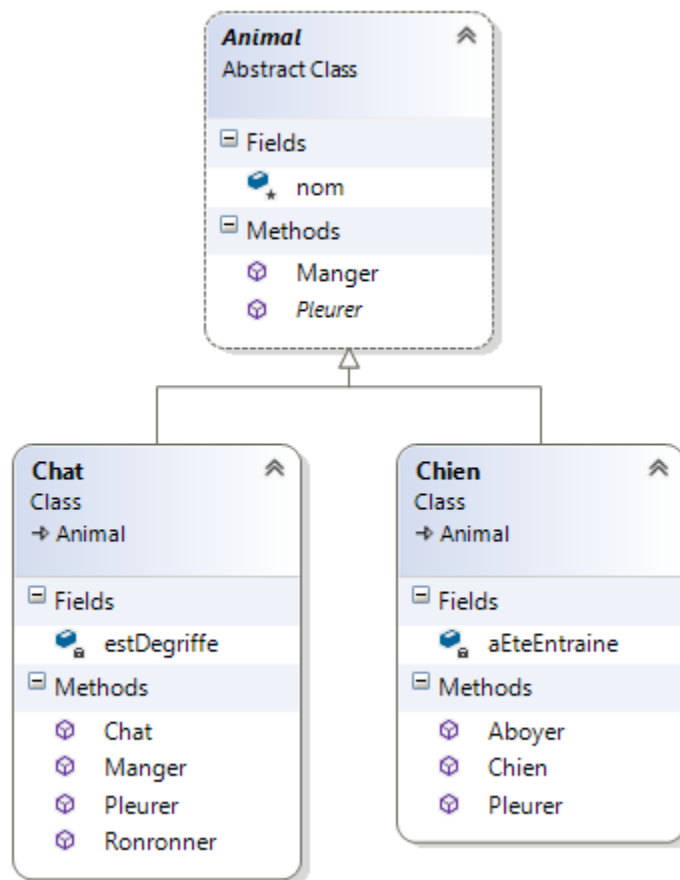
Cette dernière caractéristique des langages orientés objet concerne l'habileté à traiter des objets reliés par une relation d'héritage de façon unifiée.

Plus spécifiquement, ce principe permet à une classe de base de définir un ensemble de membres communs à tous ses descendants (classes dérivées) : cet ensemble est ce qu'on appelle une « **interface polymorphique** ».

Les membres peuvent être de nature virtuel ou abstrait :

- **Un membre virtuel** est un membre dans la classe de base qui **définit une implémentation par défaut** qui peut être changée (redéfinie) par une classe dérivée;
- **Un membre abstrait** est un membre déclaré dans la classe de base mais dont **l'implémentation n'est pas définie** par cette dernière, et où **l'implémentation doit être obligatoirement définie par toute classe dérivée**.

Si on reprend notre exemple d'animal et de chat, on peut imaginer une classe dérivée (« concurrente » à Chat), comme la classe Chien, tel que :



Grâce au polymorphisme, on pourrait avoir une collection d'animaux (dans un tableau ou une liste, par exemple) qui peut contenir des objets de n'importe quelle classe dérivée d'animaux pour lesquels on peut appeler n'importe quel membre commun entre tous les types dérivés de la classe `Animal`.

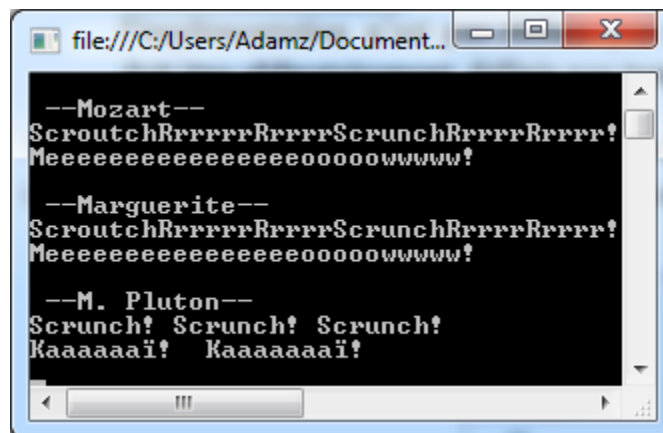
Par exemple, si on a le code suivant dans le `Main()` :

```
static void Main(string[] args)
{
    Animal[] animaux = new Animal[3];
    animaux[0] = new Chat("Mozart", true);
    animaux[1] = new Chat("Marguerite", false);
    animaux[2] = new Chien("M. Pluton", true);

    foreach (Animal a in animaux)
    {
        Console.WriteLine("\n --" + a.nom + "--");
        a.Manger();
        a.Pleurer();
    }

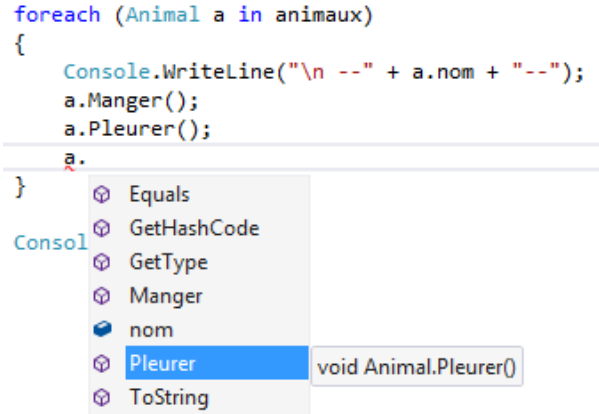
    Console.ReadLine();
}
```

On aura ceci en sortie :



Si un membre est redéfini dans une classe dérivée (comme `Manger()`, qui est redéfini dans `Chat` mais pas dans `Chien`), alors le membre appelé ne sera pas celui de la classe de base mais bien de la classe dérivée.

Évidemment, vous n'avez accès qu'aux membres communs de l'extérieur :



3.6 MODIFICATEURS D'ACCÈS

Les types (classes, interfaces, structures, énumérations et délégués) et leurs membres (propriétés, méthodes, constructeurs et champs) sont définis en utilisant un mot clé qui représente leur visibilité, c'est-à-dire **à quelle partie de l'application l'élément en question est-il accessible**. En Java, on parlera généralement de « portée ». En C#, le terme officiel est « accès ».

Ces mots clés sont ce qu'on appelle des modificateurs d'accès.

3.6.1 Public, private, protected et internal

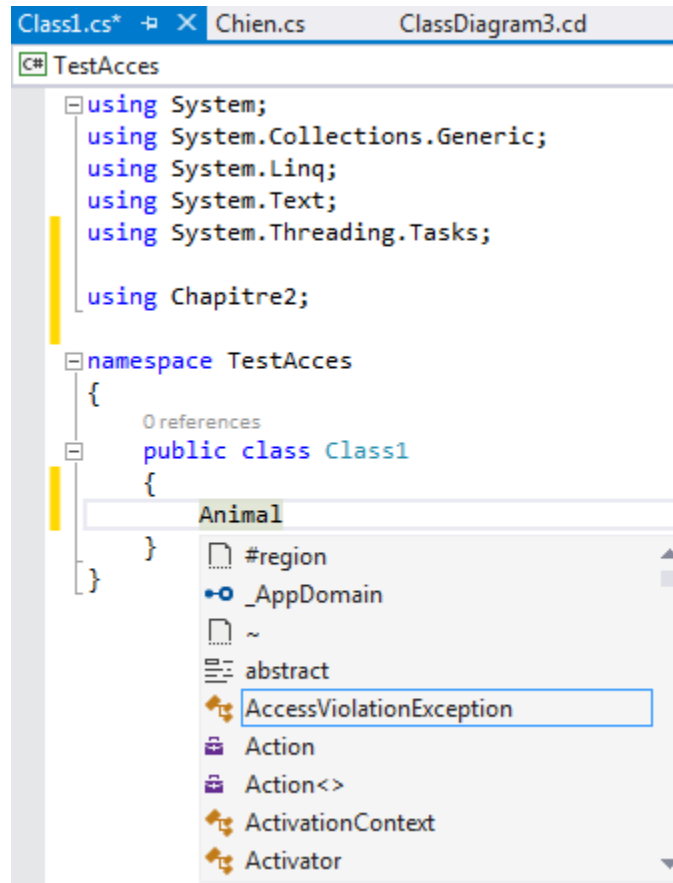
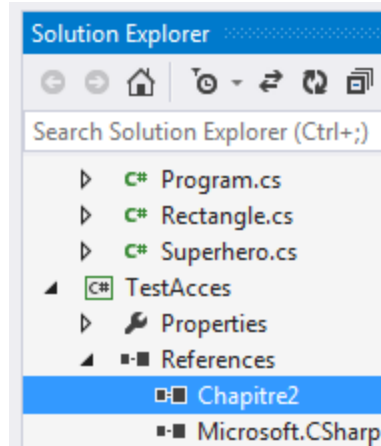
Le tableau suivant liste les différents modificateurs d'accès en C#, ce à qui ils peuvent s'appliquer (« être mis devant »), ainsi que la description de l'accès permis par ceux-ci :

Modificateur d'accès	Peut s'appliquer à	Description
public	Tout type Tout membre	Les éléments définis comme étant public n'ont aucune restriction d'accès : ils sont accessibles à l'intérieur du contexte de la classe ou à partir de toute classe externe. Ils sont accessibles depuis d'autres assembly externes (pour autant évidemment qu'il y ait une référence dans le projet vers cet assembly)
private	Tout membre	Les éléments privés sont uniquement

Modificateur d'accès	Peut s'appliquer à	Description
	Tout type imbriqué	accessibles depuis l'intérieur de la classe où la structure dans laquelle ils sont définis. Il n'est pas possible d'y accéder, par exemple, à partir d'un autre type.
<code>protected</code>	Tout membre Tout type imbriqué	Les éléments protégés peuvent être accédés seulement par la classe où ils sont définis, ainsi que par toute classe dérivée. Comme pour les éléments privés, Il n'est pas possible d'y accéder à partir d'un autre type (<i>autre que dérivé</i>).
<code>internal</code>	Tout type Tout membre	Les éléments internes sont accessibles uniquement à l'intérieur de l'assembly où ils sont définis. De ce fait, ils ne sont visibles à partir d'un assembly différent.
<code>protected internal</code>	Tout membre Tout type imbriqué	Les éléments internes protégés sont accessibles uniquement à l'intérieur de l'assembly où ils sont définis, ou par une classe dérivée définie dans un assembly externe .
(aucun)	-	Par défaut : ➤ <u>Membres</u> : implicitement privés (modificateur private); ➤ <u>Types</u> : implicitement internes (modificateur internal).

Dans ce chapitre, on utilisera uniquement `public` et `private`. Nous utiliserons `protected` à partir du prochain chapitre.

Pour le cas d'éléments définis sans modificateur d'accès, vous pouvez observer le tout si vous vous créez un nouveau projet, que vous ajoutez le premier en référence et que vous tentez de déclarer une variable d'un type (disons `Animal`) :



Aucun des types utilisés dans les démonstrations du chapitre #2 ne sont visibles, car aucun n'a le modificateur d'accès `public` et que nous nous trouvons dans un assembly différent.

3.6.2 Modificateurs de portée et types imbriqués

Comme indiqué dans le tableau précédent, les modificateurs d'accès `private`, `protected` et `protected internal` peuvent être appliqués à un type imbriqué. Nous utiliserons cela intensément au prochain chapitre, mais disons pour l'instant qu'un type imbriqué est un type déclaré directement à l'intérieur de la portée d'une classe ou d'une structure.

Si on reprend un exemple familier comme le personnage qui a une certaine classe de personnage, cela voudrait dire qu'on pourrait avoir le code suivant :

```
public class Personnage
{
    private enum ClassePersonnage {Assassin, Guerrier, Mage, Voleur}


    public Personnage() {}
}
```

Dans lequel l'énumération serait un type imbriqué qui pourrait être déclaré comme étant privé.

Toutefois, vous ne pouvez pas utiliser `private` devant des types ou des membres non imbriqués. Cela générera une erreur de compilation :

```
1 reference
private class Personnage
{
    0 references
    private enum ClassePersonnage {Assassin, Guerrier, Mage, Voleur}

    0 references
    public Personnage() {}
}
```



3.7 ENCAPSULATION

Le but de l'encapsulation étant de protéger l'intégrité des données d'une instance d'une classe, on déclarera naturellement ces champs d'instances avec le modificateur d'accès `private` (ou `protected` lorsqu'on désire une propagation vers les classes dérivées).

Pour accéder ou modifier ces données (si cela est souhaitable, car ce n'est pas toujours le cas; on peut vouloir ne pas permettre l'accès ou la modification à certaines données), on permettra de le faire indirectement via un offre de méthodes d'accès et de modifications de portée publique.

En C#, deux techniques sont possibles pour fournir de tels « privilèges » :

- Définir une paire de méthodes publiques, soit un accesseur (*get*) et un mutateur (*set*);
- Définir une propriété publique .NET, qui est un mécanisme propre à .NET.

3.7.1 Accesseurs et mutateurs

Traditionnellement (comme en Java), on voudra définir des méthodes d'accès et de mutation sur nos variables d'instances.

La convention veut généralement qu'on les nomme :

- `public TypeChamp GetNomChamp()`
- `public void SetNomChamp(TypeChamp NomChamp)`

Reprenons notre exemple de superhéro.

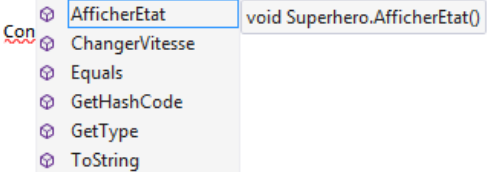
Une première étape consiste donc à rendre toutes nos variables d'instances privées (la classe `Superhero` a aussi maintenant le modificateur d'accès `public`, puisqu'on suppose qu'on pourra vouloir l'utiliser depuis un autre assembly):

```
public class Superhero
{
    //-- ces champs composent l'état d'un superhéro --
    private string nom;
    private string alias; // vrai nom
    private string superPouvoir;
    private string ligneSignature; // pour provoquer l'ennemi
    private int force;
    private int vitesse;
    private int vitesseActuelle;
    private bool peutVoler;

    public Superhero(string nom, string alias, string superPouvoir, string
        ligneSignature, int vitesse = 0, int vitesseDepart = 10, int force = 5,
        bool peutVoler = true)
    {
        this.nom = nom;
        this.alias = alias;
        this.vitesse = vitesse;
        this.vitesseActuelle = vitesseDepart;
        this.force = force;
        this.superPouvoir = superPouvoir;
        this.ligneSignature = ligneSignature;
        this.peutVoler = peutVoler;
    }
}
```

```
(...)  
}
```

Si on tente d'accéder à des données sur un superhéro, on ne pourra pas :

```
static void Main(string[] args)  
{  
    Superhero sh = new Superhero("Super Parici", "James Sergon", "Cracher du H2O", "Voyons! L'eau est bonne pour la santé!");  
    sh.  
} 
```

On n'a accès qu'aux deux méthodes publiques (`AfficherEtat()` et `ChangerVitesse()`), ainsi qu'aux méthodes fournies par `System.Object()`.

La définition d'accesseurs et de mutateurs pour quelques-uns de ces champs donnerait ceci :

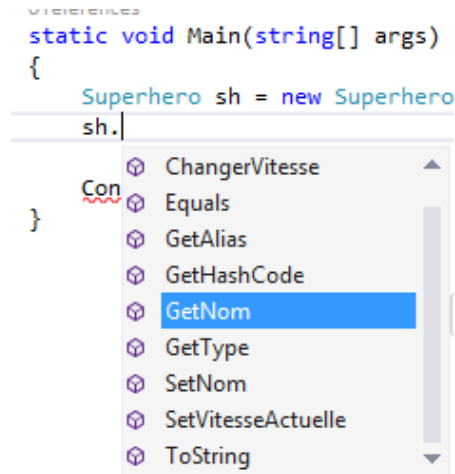
```
#region Accesseurs et mutateurs  
  
public string GetNom() { return nom; }  
public void SetNom(string nom) { this.nom = nom; }  
  
public string GetAlias() { return alias; } // seulement un accesseur pour alias  
  
public void SetVitesseActuelle(int vitesseActuelle) // seulement un mutateur, avec calculs  
{  
    if (vitesseActuelle < 0)  
        this.vitesseActuelle = 0;  
    else if (vitesseActuelle > this.vitesse)  
        this.vitesseActuelle = this.vitesse;  
    else  
        this.vitesseActuelle = vitesseActuelle;  
}  
  
#endregion
```

Il y a là un exemple d'un champ pour lequel on a un mutateur et un accesseur, un autre pour lequel on n'a qu'un accesseur mais pas de mutateur, puis un dernier pour lequel on n'a pas d'accesseur mais uniquement un mutateur qui, de plus est, doit faire certains traitements en fonction de la valeur passée en paramètre avant de muter la valeur du champ.

Note :

Remarquez l'utilisation de `#region` pour entourer les accesseurs et mutateurs. Ce n'est pas une pratique rare d'isoler ce code du reste de la classe, puisqu'il s'agit souvent de code appelé à rester inchangé une fois défini.

Si on désire accéder aux mutateurs et/ou accesseurs de l'extérieur, on fait évidemment comme pour toute autre méthode. :



3.7.2 Propriétés

Bien qu'elle respecte tout à fait le principe d'encapsulation, cette façon de faire n'est toutefois pas idéale en C#. En fait, en .NET, on met à votre disposition la notion de **propriétés** (.NET) comme méthode officielle d'encapsulation (d'accès et de mutation). Ils agissent en quelque sorte comme une simplification (et une distinction) par rapport aux accesseurs et mutateurs traditionnels.

3.7.2.1 UTILISATION ET SYNTAXE GÉNÉRALE

Voici les méthodes précédentes changées en propriétés :

```
public class Superhero
{
    //-- ces champs composent l'état d'un superhéro --
    private string nom;
    private string alias; // vrai nom
    private string superPouvoir;
    private string ligneSignature; // pour provoquer l'ennemi
    private int force;
    private int vitesse;
    private int vitesseActuelle;
}
```

```
private bool peutVoler;

#region Propriétés (accesseurs et mutateurs)

public string Nom
{
    get { return this.nom; }
    set { this.nom = value; }
}

public string Alias
{
    get { return this.alias; }
    set { this.alias = value; }
}

public int VitesseActuelle
{
    get { return this.vitesseActuelle; }
    set
    {
        if (value < 0)
        {
            this.vitesseActuelle = 0;
        }
        else if (vitesseActuelle > this.vitesse)
        {
            this.vitesseActuelle = this.vitesse;
        }
        else
        {
            this.vitesseActuelle = value;
        }
    }
}
#endregion
(...)
```

Pour l'instant, les trois propriétés ont toutes un accesseur et un mutateur défini.

Observez et comprenez les éléments suivants :

- Pour déclarer une propriété, il suffit donc de donner le type, puis, par convention, la nommer en utilisant le nom du champ avec une capitalisation de la première lettre;
- À l'intérieur, il faut alors utiliser les mots clés `get` et `set` et donner les instructions d'accès et de mutation :
 - Vous pouvez avoir des traitements particuliers avant de retourner ou de modifier le champ associé à la propriété;
 - Dans le cas du `set`, remarquez le jeton `value`, qu'on appelle un « mot clé contextuel » : il représente toujours la valeur affectée à la propriété (dans un véritable sens d'affectation)

Ainsi, du côté appelant, plutôt que d'avoir quelque chose comme ceci :

```
static void Main(string[] args)
{
    Superhero sh = new Superhero("Super Parici", "James Sergon", "Cracher
    du H2O", "Voyons! L'eau est bonne pour la santé!");
    sh.SetNom("Super Sacabou");
    sh.SetVitesseActuelle(5);
    Console.WriteLine(sh.GetAlias());
    sh.AfficherEtat();

    Console.ReadLine();
}
```

On aurait ceci :

```
static void Main(string[] args)
{
    Superhero sh = new Superhero("Super Parici", "James Sergon", "Cracher
    du H2O", "Voyons! L'eau est bonne pour la santé!");
    sh.Nom = "Super Sacabou";
    sh.VitesseActuelle = 5;
    Console.WriteLine(sh.Alias);
    sh.AfficherEtat();

    Console.ReadLine();
}
```

C'est ainsi moins chargé et plus naturel par rapport aux accesseurs et mutateurs classiques.

Par ailleurs, ils rendent certaines autres manipulations plus simples. Par exemple, imaginons qu'on souhaite incrémenter la vitesse de 1. Traditionnellement, il faudrait utiliser en conjoncture un mutateur qui appelle un accesseur, dans le style qui suit :

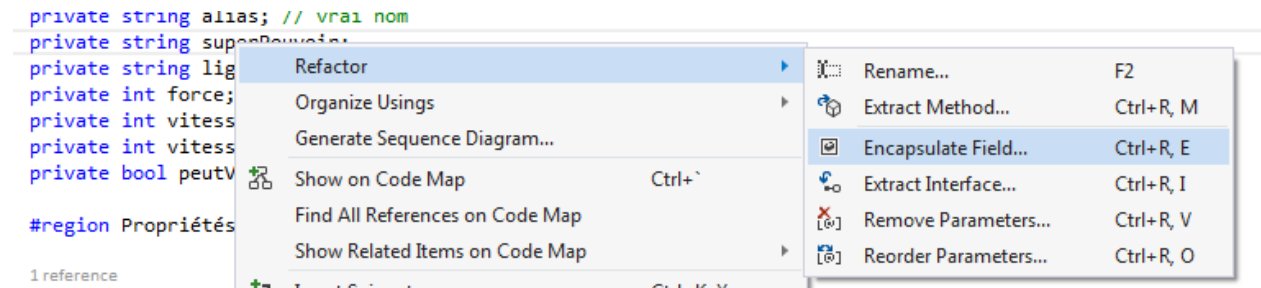
```
sh.SetVitesseActuelle(sh.GetVitesseActuelle() + 1);
```

Avec les propriétés, il suffit de faire ceci :

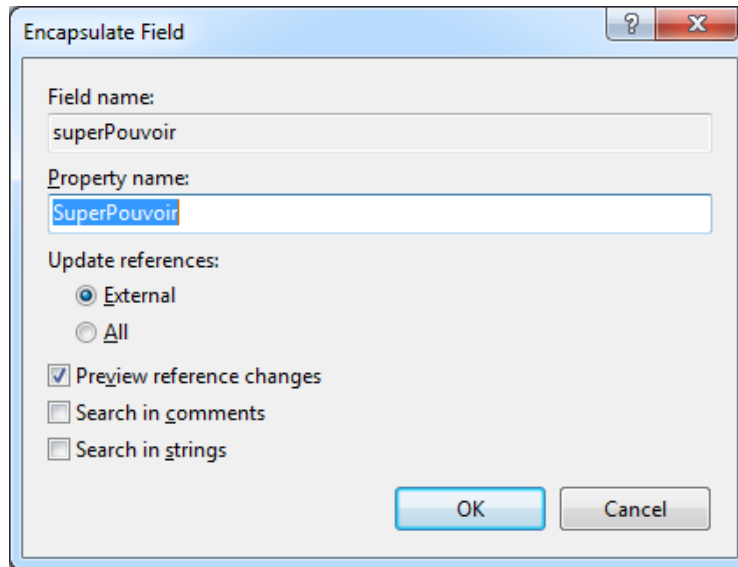
```
sh.VitesseActuelle++;
```

Et le tour est joué!

Enfin, vous pouvez générer des squelettes de propriétés automatiquement pour vos champs avec Visual Studio. Il suffit de sélectionner le champ en question, puis d'appuyer sur le clic droit de la souris, et se diriger dans le menu contextuel vers *Refactor > Encapsulate Field...* :



On vous demandera de spécifier le nom de la propriété (le nom proposé par défaut correspond à la convention mentionnée plus tôt) :



Le code sera alors généré pour vous :

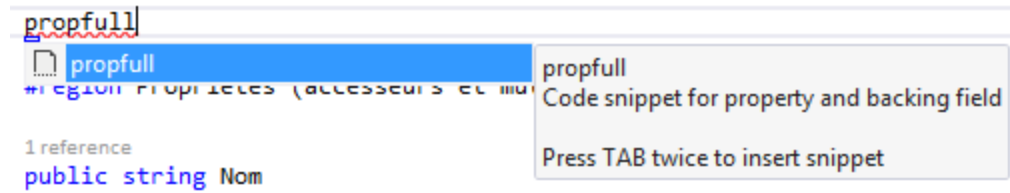
```
private string alias; // vrai nom
private string superPouvoir;

0 references
public string SuperPouvoir
{
    get { return superPouvoir; }
    set { superPouvoir = value; }
}

private string ligneSignature; // pour f
```

Vous pouvez aussi utiliser des *snippets* pour vous assister dans le design de propriétés.

Par exemple, si vous tapez `propfull` et appuyez à deux reprises sur `tab` :



Ceci sera créé pour vous :

```
private int myVar;

- references
public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}
```

Un autre aspect des propriétés est leur utilisation à l'intérieur de la définition d'une classe.

Les propriétés peuvent servir pour spécifier des règles ou des contraintes sur les valeurs des champs. Le fait que la vitesse actuelle doit être comprise entre 0 et la vitesse maximale représente une telle règle, et cela est capturé dans la propriété `VitesseActuelle` telle que précédemment définie.

Lorsqu'on fait appel au constructeur de la classe `Superhero`, il faudrait aussi tenir compte de cette règle.

Plutôt que de réimplémenter cette règle dans le constructeur (redondance!) comme suit :

```
public Superhero(string nom, string alias, string superPouvoir, string
ligneSignature, int vitesse = 0, int vitesseDepart = 10, int force = 5,
bool peutVoler = true)
{
    this.nom = nom;
    this.alias = alias;
    this.vitesse = vitesse;

    if (vitesseDepart < 0)
        this.vitesseActuelle = 0;
    else if (vitesseActuelle > this.vitesse)
        this.vitesseActuelle = this.vitesse;
    else
        this.vitesseActuelle = vitesseDepart;

    this.force = force;
}
```

```
        this.superPouvoir = superPouvoir;  
        this.ligneSignature = ligneSignature;  
        this.peutVoler = peutVoler;  
    }
```

On peut utiliser directement la règle pré-implémentée dans notre propriété qui encapsule notre champ `vitesseActuelle` :

```
public Superhero(string nom, string alias, string superPouvoir, string  
ligneSignature, int vitesse = 0, int vitesseDepart = 10, int force = 5,  
bool peutVoler = true)  
{  
    this.nom = nom;  
    this.alias = alias;  
    this.vitesse = vitesse;  
    this.VitesseActuelle = vitesseDepart;  
    this.force = force;  
    this.superPouvoir = superPouvoir;  
    this.ligneSignature = ligneSignature;  
    this.peutVoler = peutVoler;  
}
```

Note :

De façon générale, il est recommandable de faire appel aux propriétés encapsulantes chaque fois que le champ encapsulé est susceptible d'être révisé par des règles particulières. Cela a pour but de favoriser la maintenabilité du code.

3.7.2.2 PROPRIÉTÉS EN LECTURE ET ÉCRITURE SEULE

Pour avoir des propriétés en lecture ou en écriture seule, il suffit d'omettre un bloc `get` ou `set`.

Par exemple :

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

Ou

```
public string Nom  
{  
    set { this.nom = value; }  
}
```

3.7.2.3 PROPRIÉTÉS STATIQUES

Vous pouvez aussi définir des propriétés statiques sur des champs de classe (statiques). La syntaxe générale reste identique, mise à part le fait que vous devez mettre le mot clé `static` devant le type de la propriété.

Si on reprend notre exemple de compte bancaire, on pourrait donc encapsuler notre classe en déclarant la balance comme étant privée et en lui attribuant une propriété, puis faire de même pour le champ statique.

On se retrouverait donc avec le code suivant :

```
class CompteBancaire
{
    private static double tauxRistourne;
    private double balance;

    public static double TauxRistourne
    {
        get { return tauxRistourne; }
    }

    public double Balance
    {
        get { return this.balance; }
        set { this.balance = value; }
    }

    (...)
}
```

D'un point de vue extérieur, on aura donc accès plus proprement à ces champs :

```
static void Main(string[] args)
{
    Console.ReadLine()
    CompteBancaire.
    {
        DecrementerRistourne
        Equals
        IncrementerRistourne
        ReferenceEquals
        TauxRistourne
    }
}
```

double CompteBancaire.TauxRistourne

3.7.2.3 PROPRIÉTÉS AUTOMATIQUES

Prenons l'exemple d'une classe `Joueur` définie telle que ceci :

```
public class Joueur
{
```

```
private string nom;  
private string prenom;  
private int numero;  
  
public Joueur(string nom, string prenom, int numero)  
{  
    this.nom = nom;  
    this.prenom = prenom;  
    this.numero = numero;  
}  
  
public string Nom  
{  
    get { return this.nom; }  
    set { this.nom = value; }  
}  
  
public string Prenom  
{  
    get { return this.prenom; }  
    set { this.prenom = value; }  
}  
  
public int Numero  
{  
    get { return this.numero; }  
    set { this.numero = value; }  
}  
}
```

On peut également avoir des propriétés qui ne sont pas directement reliées 1@1 à une variable d'instance, comme ceci :

```
public string NomComplet  
{  
    get { return this.prenom + " " + this.nom; }  
}
```

Cette syntaxe – vous serez d'accord – est pratique et claire, mais dans tous les cas où, comme les propriétés `Nom`, `Prenom` et `Numero`, on souhaite uniquement avec un accès en lecture et en écriture à un champ, cela fait beaucoup de code qui n'apporte aucun nouveau traitement dans la classe ou la structure. On peut imaginer que si on avait 12 champs de données, il faudrait créer 12 blocs de propriétés (si on souhaite avec un quelconque accès de l'extérieur sur l'ensemble de ces champs).

C# permet d'alléger le code relatif aux propriétés grâce aux propriétés automatiques qui permettent pour ces champs de déterminer directement à la déclaration qu'on souhaite créer une propriété liée.

Par exemple, en reprenant cette classe `Joueur`, on aurait alors plutôt le code suivant :

```
public class Joueur
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int Numero { get; set; }

    public Joueur(string nom, string prenom, int numero)
    {
        this.Nom = nom;
        this.Prenom = prenom;
        this.Numero = numero;
    }
}
```

Lorsque vous définissez de telles propriétés automatiques, comme pour des propriétés régulières, vous devez les déclarer comme étant publiques, donner le type de données, attribuer un nom et indiquer seulement `get;` et `set;` (corps vide).

À la compilation, cette propriété automatique autogénèrera un champ privé pour la classe ou la structure et la logique relative à l'accesseur et au mutateur.

Cependant, contrairement aux propriétés régulières, vous ne pouvez pas définir des propriétés automatiques en lecture seule ou en écriture seule. Ces cas-ci causeraient des erreurs à la compilation :

```
public bool EstBlesse { get; }
public bool EstEnMomentum { set; }
```

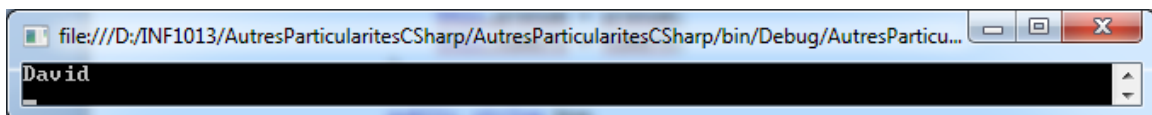
Pour utiliser les propriétés du côté d'une méthode cliente, cela se déroulera comme pour une propriété régulière. Par exemple :

```
static void Main(string[] args)
{
    Joueur j = new Joueur("Adam", "Joly", 2);
    j.Nom = "David";

    Console.WriteLine(j.Nom);

    Console.ReadLine();
}
```

On aura évidemment en sortie :



Lorsque vous utilisez des propriétés automatiques pour encapsuler des données primitives ou de types de référence, vous pouvez alors utiliser immédiatement après l'instanciation

de la structure ou de la classe étant donné que les champs d'instances autogénérés se voient automatiquement attribués une valeur par défaut (0 ou `null`).

Lorsque vous déclarez des champs d'instances, normalement vous pouvez affecter directement une valeur « par défaut » qui est différente (autre que 0 ou `null`) directement dans la ligne de la déclaration. Par exemple :

```
public class Joueur
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int Numero { get; set; }
    private bool estMarcheAutonome = true;

    public Joueur(string nom, string prenom, int numero)
    {
        this.Nom = nom;
        this.Prenom = prenom;
        this.Numero = numero;
    }
}
```

Si vous désirez avoir une valeur d'instanciation différente pour vos propriétés automatiques, vous n'aurez pas d'autres choix que de les définir à l'intérieur d'un constructeur, comme ceci :

```
public class Joueur
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int Numero { get; set; }
    public bool EstMarcheAutonome { get; set; } // mis d'abord à false

    public Joueur(string nom, string prenom, int numero)
    {
        this.Nom = nom;
        this.Prenom = prenom;
        this.Numero = numero;
        EstMarcheAutonome = true;
    }
}
```

On voit donc que les propriétés automatiques sont très utiles pour encapsuler rapidement et clairement des champs de classes ou de structures, c'est-à-dire sans encombrer les modules de code non nécessaire.

Gardez en tête que si vous devez définir une propriété qui nécessite du code additionnel (implémenter des règles) pour déterminer ou recueillir la valeur (validation de valeur, communication avec une base de données, fusion partielle ou totale de certaines valeurs

d'autres champs, ...) ou que cette propriété doit être en lecture ou en écriture seule, vous aurez alors intérêt à utiliser une propriété régulière.

3.8 INITIALISATION D'OBJETS

Depuis (encore) la version C# .NET 2008, une nouvelle syntaxe est mise à la disposition des programmeurs afin d'injecter directement des valeurs aux champs des structures en passant par des propriétés (ou bien si les champs sont publiques/non encapsulés) des structures ou des classes sans passer par les instructions d'un constructeur.

Par exemple, prenons une classe nommée `Point3D` avec des propriétés automatiques pour les variables `x`, `y` et `z` :

```
public class Point3D
{
    public float X { get; set; }
    public float Y { get; set; }
    public float Z { get; set; }
}
```

Classiquement, pour initialiser les valeurs pour `x`, `y` et `z`, on procéderait ainsi :

```
static void Main(string[] args)
{
    Point3D p = new Point3D();
    p.X = 4.0f;
    p.Y = 2.0f;
    p.Z = -1.0f;
}
```

Ou, si on avait tout de même défini un constructeur personnalisé tel que :

```
public class Point3D
{
    public float X { get; set; }
    public float Y { get; set; }
    public float Z { get; set; }

    public Point3D(float x, float y, float z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }
}
```

On aurait pu instancier comme ceci :

```
static void Main(string[] args)
{
    Point3D p = new Point3D(4.0f, 2.0f, -1.0f);
}
```

3.8.1 Description

Avec la nouvelle syntaxe d'initialisation d'objets, vous pouvez écrire plutôt ceci :

```
Point3D p = new Point3D { X = 4.0f, Y = 2.0f, Z = -1.0f };
```

Sans avoir besoin de définir un constructeur dans la structure ou la classe.

Ce faisant, en arrière-scène, le constructeur par défaut de la classe ou la structure est d'abord appelé, puis le mutateur de chacune des propriétés spécifiées dans la syntaxe d'initialisation sont appelées une à la suite de l'ordre.

La façon présentée appelle donc implicitement le constructeur par défaut dans l'exemple précédent. Cependant, sachez que vous pouvez explicitement déterminer un constructeur à appeler avant de faire appel aux propriétés.

En d'autres termes, ceci fonctionnerait également (ce serait alors une syntaxe explicite) :

```
Point3D p = new Point3D() { X = 4.0f, Y = 2.0f, Z = -1.0f };
```

Vous pourriez même en fait utiliser explicitement n'importe quel des constructeurs définis pour la classe ou la structure. Par exemple, ceci serait également valide :

```
Point3D p = new Point3D(1.0f, 0.0f, 5.0f) { X = 4.0f, Y = 2.0f, Z = -1.0f };
```

Comme l'appel aux propriétés se ferait alors après l'appel au constructeur, les coordonnées du point seraient (4, 2, -1) et non (1, 0, 5).

Évidemment, il y a peu de raison de vouloir faire appel à un initiateur d'objet dans ce dernier cas, mais imaginons que notre point a également une autre variable d'instance comme une couleur et un constructeur qui prend une couleur comme paramètre :

```
public enum CouleurPoint { Noir, Rouge, Bleu, Orange, Vert, Gris, Blanc }

public class Point3D
{
    public float X { get; set; }
    public float Y { get; set; }
    public float Z { get; set; }
    private CouleurPoint couleur;
```

```
    public Point3D(CouleurPoint couleur)
    {
        this.couleur = couleur;
    }
}
```

L'utilisation d'un initialiseur d'objet devient alors plus appropriée :

```
Point3D p = new Point3D(CouleurPoint.Gris) { X = 4.0f, Y = 2.0f, Z = -1.0f };
```

3.8.2 Initialisation de champs « internes »

Imaginons maintenant la classe `PrismeRectangulaire` pour laquelle on a deux champs privées :

```
public classe PrismeRectangulaire
{
    private Point3D coinSuperieurGauche;
    private Point3D coinInferieurDroite;
}
```

Comme vous devriez le savoir, on peut instancier ces types directement lors de la déclaration en faisant appel à leur constructeur (ce que vous ne pouvez pas faire avec une propriété automatique!):

```
public class PrismeRectangulaire
{
    private Point3D coinSuperieurGauche = new Point3D(CouleurPoint.Blanc);
    private Point3D coinInferieurDroite = new Point3D(CouleurPoint.Gris);
}
```

Vous pourriez alors initialiser ces champs :

```
public class PrismeRectangulaire
{
    private Point3D coinSuperieurGauche = new Point3D(CouleurPoint.Noir) { X =
0.0f, Y = 0.0f, Z = 0.0f };
    private Point3D coinInferieurDroite = new Point3D(CouleurPoint.Noir) { X =
1.0f, Y = 1.0f, Z = 1.0f };
}
```

On pourrait ainsi signifier succinctement que par défaut le prisme est noir et qu'il a une dimension de 1 x 1 x 1 à partir du point d'origine dans l'espace.

Ajoutons maintenant des propriétés pour ces champs :

```
public class PrismeRectangulaire
{
    private Point3D coinSuperieurGauche = new Point3D(CouleurPoint.Noir) { X =
0.0f, Y = 0.0f, Z = 0.0f };
```

```
private Point3D coinInferieurDroite = new Point3D(CouleurPoint.Noir) { X = 1.0f, Y = 1.0f, Z = 1.0f };
```

```
public Point3D CoinSuperieurGauche
{
    get { return this.coinSuperieurGauche; }
    set { this.coinSuperieurGauche = value; }
}

public Point3D CoinInferieurDroite
{
    get { return this.coinInferieurDroite; }
    set { this.coinInferieurDroite = value; }
}
}
```

En utilisant la syntaxe d'initialisation d'objet, vous pouvez alors aussi spécifier directement des instances pour chacun des champs publics ou, comme dans ce cas, des propriétés liées à des champs d'instance privées :

```
static void Main(string[] args)
{
    PrismeRectangulaire prisme = new PrismeRectangulaire
    {
        CoinSuperieurGauche = new Point3D(CouleurPoint.Bleu) { X = 0.0f, Y = 1.0f, Z = -1.5f },
        CoinInferieurDroite = new Point3D(CouleurPoint.Rouge) { X = 2.0f, Y = 2.0f, Z = -2.5f }
    };
}
```

Bref, on peut imbriquer l'initialisation d'objet.

3.9 CONSTANTES

Vous pouvez définir des constantes, c'est-à-dire des données pour lesquelles la valeur ne change plus après l'affectation initiale. C'est particulièrement utile pour définir un ensemble de valeurs connues qui servira ensuite dans des calculs pour les types auxquels elles sont liées.

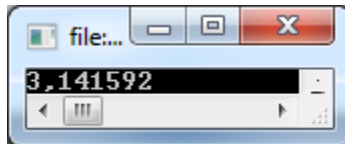
Il suffit pour cela d'utiliser le mot clé `const`. **Les constantes sont implicitement statiques.**

Par exemple :

```
class MaClasse
{
    public const double PI = 3.141592;
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MaClasse.PI);
        Console.ReadLine();
    }
}
```

On aurait :



Toutefois, on aurait une erreur si on avait écrit ceci :

```
public class MaClasse
{
    public const double PI = 3.141592;
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        MaClasse.PI = 4;
        class Chapitre2.MaClasse
    }
}

Error:
The left-hand side of an assignment must be a variable, property or indexer
```

Ou cela, comme quoi l'affectation doit obligatoirement être faite dès la déclaration (c'est-à-dire déterminée à la compilation) :

```
public class MaClasse
{
    public const double PI;
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        MaClasse.PI = 3.141592;
        Console.WriteLine(MaClasse.PI);
    }
}
```

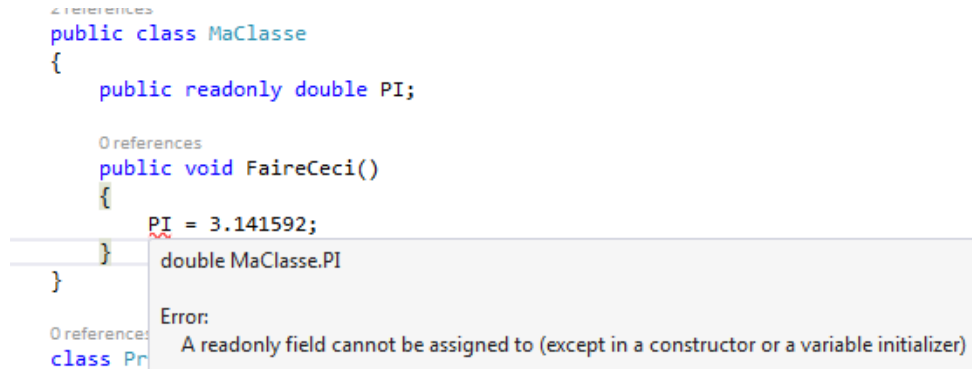
Si vous voulez retarder l'affectation de la valeur à l'exécution (*parce que cette valeur dépendra du contexte d'exécution ou de la valeur d'autres données, par exemple*), vous trouverez aussi en C# des champs de données en **lecture seule**. Le code suivant deviendrait donc valide :

```
public class MaClasse
{
    public readonly double PI;

    public MaClasse()
    {
        PI = 3.141592;
    }
}
```

La seule condition est que **la valeur du champ en lecture seule doit être affectée dans un constructeur uniquement**. Cela ne fonctionnera pas ailleurs.

Par exemple, ceci génère une erreur :



Mais ceci est légal :

```
public class MaClasse
{
    public readonly double PI = 3.141592;

    public MaClasse()
    {
        PI = 3.15;
        PI = 3.2;
    }

    public MaClasse(int a)
    {
        PI = 3.13 * a;
    }
}
```

Vous pouvez donc préaffecter une valeur, mais vous laissez la possibilité de la modifier à plusieurs reprises dans un même constructeur, ou dans des constructeurs différents.

Les champs en lecture seule ne sont cependant pas implicitement statiques. Si vous voulez qu'un champ en lecture seule soit au niveau de la classe et non d'une instance, vous devrez utiliser le mot clé `static` usuel :

Par exemple :

```
class MaClasse
{
    public static readonly double PI = 3.141592;
}

class Program
```

```
{
    static void Main(string[] args)
    {
        Console.WriteLine(MaClasse.PI);
        Console.ReadLine();
    }
}
```

Et à nouveau, la seule façon d'affecter une valeur à ce champ autre part que directement à l'affectation est par un constructeur statique :

```
class MaClasse
{
    public static readonly double PI = 3.141592;

    public static MaClasse()
    {
        PI = 3;
    }
}
```

3.10 TYPES PARTIELS

En C#, il y a également ce qu'on appelle les types partiels. En fait, C# vous offre la possibilité de séparer le code d'une classe à travers plusieurs fichiers .cs différents si vous le souhaitez.

Si tel est votre souhait, alors à partir du moment où une classe est définie à travers plusieurs blocs (normalement dans plusieurs fichiers différents), **vous devez absolument utiliser le mot clé `partial` devant tous les noms des classes concernées.**

Par exemple :

```
partial class MaClasse
{
    // constructeurs

    // méthodes
}

partial class MaClasse
{
    // champs

    // propriétés
}
```


Si vous avez déjà travaillé avec les Windows Forms, vous avez peut-être remarqué que cette notion est largement utilisée.

Par exemple, dans un projet Windows Forms créé par défaut, vous trouvez dans *Form1.cs* le code suivant :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Et le code suivant dans *Form1.Designer.cs* :

```
partial class Form1
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resources should be disposed;
    otherwise, false.</param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code
```

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}

#endregion

}
```