

# Les classes et les structures

L'utilisation d'objets étant au cœur de la POO, nous commencerons par voir comment les créer, de la déclaration de la classe ou de la structure qui servira de modèle, jusqu'à l'instanciation. Nous verrons aussi comment ajouter des propriétés et des fonctionnalités à nos objets.

## 1. Classes

La majorité des types définis dans le framework .NET sont des classes, aussi il est très important de comprendre comment les manipuler. En effet, toute application écrite en C# comportera au minimum une classe écrite par le développeur, et utilisera probablement des dizaines/centaines de classes issues du framework .NET.

### a. Déclaration

Une classe se définit et est utilisée au travers de son nom. Il faut respecter certaines règles pour ce nommage, sous peine de se voir dans l'impossibilité de compiler. Les possibilités d'utilisation de la classe sont quant à elles définies par le modificateur d'accès associé.

#### Nom d'une classe

Le nom d'une classe n'est valide que s'il respecte les règles suivantes :

- Il ne contient que des caractères alphanumériques ou le caractère \_.
- Il ne commence pas par un chiffre.

Par convention, il est d'usage de nommer les classes en respectant le style "UpperCamelCase" (aussi appelé PascalCase), c'est-à-dire que la première lettre de chaque mot composant le nom de la classe doit être en majuscule, tandis que le reste est écrit en minuscules. Toutes les classes du framework .NET respectent cette convention.

MaClasse, MurEnBrique ou PorteManteau sont des noms respectant la casse "UpperCamelCase".

ecranPlat est un nom accepté mais ne respectant pas cette convention.

#### Syntaxe

Une classe est déclarée en utilisant le mot-clé `class` suivi d'un nom et d'un bloc de code délimité par les caractères `{` et `}`.

La syntaxe générale de déclaration d'une classe est la suivante :

```
<modificateur d'accès> [partial] class <nom de la classe> :  
[<Classe de base>, <Interface1>, <Interface2>, ...]  
{  
}  
}
```



Les notions de classe de base et d'interface seront étudiées dans ce chapitre aux sections L'héritage et Les interfaces.

Les classes sont des types références, ce qui signifie que les variables dont le type est une classe ont pour valeur

par défaut `null`. Ce comportement est important à connaître car il implique qu'il faut instancier un objet pour initialiser la variable. La notion d'instanciation est étudiée un peu plus loin dans ce chapitre (cf. section Utilisation des classes et structures).

## **Modificateurs d'accès**

Les modificateurs d'accès sont des mots-clés définissant la visibilité de la classe pour le reste du code. Les différents modificateurs d'accès disponibles pour les classes sont les suivants, du plus restrictif au moins restrictif :

- `private` : ce modificateur n'est autorisé que dans le cas de classes imbriquées, la classe étant dans ce cas uniquement visible par le type qui la contient (voir l'exemple ci-dessous).
- `protected` : ce modificateur n'est autorisé que dans le cas de classes imbriquées, la classe étant dans ce cas visible par le type qui la contient et par ses types dérivés (voir l'exemple ci-dessous).
- `internal` : la classe est visible par toutes les classes définies dans l'assembly.
- `protected internal` : la classe est visible par toutes les classes définies dans l'assembly et visible par une classe extérieure à l'assembly uniquement si cette dernière en hérite.
- `public` : la classe est visible par n'importe quel autre type, dans n'importe quel assembly.



Lorsqu'aucun modificateur d'accès n'est spécifié dans la déclaration, le compilateur marque la classe comme étant `internal`.

Exemple de classe imbriquée avec visibilité `private` :

```
public class Voiture
{
    private class Transmission
    {
    }
}
```

Dans le cas décrit ci-dessus, la classe `Transmission` ne sera utilisable que par `Voiture`. Si le modificateur est remplacé par `protected`, `Transmission` devient accessible par `Voiture` et ses classes filles uniquement.

## **Variables d'instance**

Les classes représentent des ensembles cohérents de données et d'actions. Il est donc possible de déclarer des variables dans les classes. On les appelle variables membres, ou membres d'instances. On les déclare de la même manière que les variables d'une fonction, à l'exception du fait qu'on peut leur assigner une visibilité.

Par défaut, si aucun modificateur d'accès n'est spécifié pour une variable, le compilateur considère que la variable est marquée comme `private`.

```
public class Voiture
{
    //Déclaration d'une variable membre publique
    public int kilometrage;
}
```

Pour utiliser une variable d'instance, il suffit de préfixer le nom de la variable par le nom de l'objet sur lequel vous voulez agir et l'opérateur point (.). Ici, pour modifier le kilométrage de notre voiture, on peut écrire :

```
//maFerrari est évidemment de type Voiture  
maFerrari.kilometrage = 42;
```

## b. Constructeur et destructeur

Tout comme les objets physiques, les objets utilisés en C# ont un cycle de vie défini. Celui-ci commence par l'instanciation de l'objet et se termine par son nettoyage par le Garbage Collector. Pour exécuter des instructions pendant chacune de ces deux opérations, il existe deux méthodes spécifiques : le constructeur et le destructeur.

### Constructeur

Un constructeur est une méthode spéciale permettant l'instanciation d'un objet. Son nom est identique à celui de la classe dans laquelle il est défini, et il n'a aucun type de retour.

Toutes les classes possèdent au moins un constructeur (à l'exception des classes abstraites que nous verrons à la section L'héritage), qu'il soit défini dans le code ou non. Si aucun constructeur n'est déclaré explicitement, le compilateur crée un constructeur vide n'acceptant aucun paramètre appelé constructeur par défaut.

Il est très souvent utile de définir un constructeur afin d'initialiser des variables membres de la classe.

La syntaxe permettant d'écrire un constructeur est la suivante :


```
<modificateur d'accès> <Nom de la classe>([<paramètre1>,  
<paramètre2>...])  
{  
}
```

Il est possible de surcharger les constructeurs, ce qui permet d'initialiser l'objet de plusieurs manières différentes en fonction du contexte d'utilisation.

```
public class Employe  
{  
    private int age;  
    private int salaire  
  
    public Employe()  
    {  
    }  
  
    public Employe(int ageEmploye)  
    {  
        age = ageEmploye;  
    }  
  
    public Employe(int ageEmploye, int salaireEmploye)  
    {  
        age = ageEmploye;  
    }  
}
```

```
        salaire = salaireEmploye;
    }
}
```

Il est possible de déclarer un ou plusieurs constructeurs avec les modificateurs d'accès `private` ou `protected`. Ceci peut s'avérer très utile lorsqu'il est nécessaire de maîtriser complètement les instanciations d'une classe, comme dans le cas de l'implémentation du patron de conception Singleton.

 Le patron de conception Singleton permet de s'assurer que le nombre maximum d'instances d'une classe est à tout moment 1.

### **Destructeur (ou finaliseur)**

La destruction d'un objet est une opération automatique dans une application .NET. Le Garbage Collector (GC) surveille la totalité des instances créées tout au long de l'exécution de l'application, et marque comme orphelin chaque objet qui n'est plus utilisé. Lorsque le système a besoin de mémoire, le GC détruit les objets qu'il considère comme nettoyables en appelant leurs destructeurs s'ils en ont un.

Un destructeur est donc une autre méthode particulière dont la signature est imposée. Son nom est le même que celui de la classe précédé par le caractère `~`. De plus, il ne peut accepter aucun paramètre et, comme le constructeur, n'a pas de type de retour.


Cette méthode, si elle est implémentée, doit exécuter le code nécessaire à la libération d'éventuelles ressources non managées qui n'auraient pas été libérées précédemment, comme des fichiers ou des connexions à des bases de données.

Le destructeur de notre classe `Voiture` peut s'écrire ainsi :

```
~Voiture()
{
    //Insérer ici le code permettant de nettoyer
    //les ressources utilisées par l'objet
}
```

Il est impossible de prévoir quand le GC détruira un objet particulier, mais il est toutefois possible de lui demander d'effectuer une collecte des objets inutilisés et de libérer la mémoire qui peut l'être. Cette opération est coûteuse en termes de ressources pour un gain qui peut être très faible (de quelques ko à plusieurs dizaines de Mo en fonction du contexte). Pour lancer cette collecte, il suffit d'exécuter la méthode statique `Collect` de la classe GC :

```
GC.Collect();
```

 Pour connaître le nombre d'octets alloués à une application en mémoire, il est possible d'utiliser le code suivant :

```
long memoireUtilisee = GC.GetTotalMemory(false);
```

L'imprédictibilité du GC pousse donc les développeurs à isoler le code de nettoyage dans une méthode appelée explicitement quand l'objet n'est plus utilisé.

Cette méthode s'appelle en général `Dispose` et fait partie de l'implémentation du patron de conception **Disposable**.

```
public void Dispose()
{
    //Code de libération des ressources
}
```

L'appel de cette méthode pouvant être omis par le développeur utilisant la classe, il convient de l'appeler à partir du destructeur afin d'être certain que la ressource sera libérée.

Un problème peut alors se poser : la méthode `Dispose` peut être appelée une fois explicitement dans le code et une fois implicitement via le destructeur. Il faut donc mettre en place un mécanisme permettant d'exécuter plusieurs fois la méthode `Dispose` sans causer d'erreur.

```
public class Voiture
{
    private bool estNettoye = false;

    public void Dispose()
    {
        if (!estNettoye)
        {
            //Code de nettoyage
            estNettoye = true;
        }
    }

    ~Voiture()
    {
        Dispose();
    }
}
```

Une autre solution est d'indiquer au GC qu'il n'est plus nécessaire d'appeler le destructeur de l'objet une fois qu'il a été nettoyé.

```
public void Dispose()
{
    //Code de libération des ressources

    GC.SuppressFinalize(this);
}
```

### c. Classes partielles

La version 2 du langage C# (arrivée avec .NET 2.0) a introduit la notion de classes partielles afin de mieux gérer la génération automatique de code. En effet, les interfaces graphiques Windows Forms sont entièrement créées à l'aide de code C#. La présence de code généré et de code écrit par les développeurs dans le même fichier était

donc source de problèmes. Avec les classes partielles, il est possible de scinder une classe en plusieurs portions pouvant être réparties dans plusieurs fichiers.

Il est ainsi possible de générer le code d'une interface graphique Windows Forms ou de classes d'accès aux données tout en laissant la possibilité d'étendre ce code sans créer de nouvelles classes ou entraîner des situations de conflits entre le code généré et le code du développeur.

On déclare une classe partielle de la même manière qu'une classe classique, mais en préfixant le mot-clé `class` par le mot-clé `partial`. Toutes les parties d'une classe partielle doivent être déclarées avec le même nom dans le même assembly et dans le même espace de noms. Si ces conditions ne sont pas respectées, le compilateur considérera les différentes parties comme des classes différentes, ce qui n'est évidemment pas l'effet attendu.

```
public partial class Employe
{
    public int salaire;
}

public partial class Employe
{
    public int age;
}
```

À la compilation, ces deux déclarations seront fusionnées. Ainsi, il sera possible d'écrire :

```
//La variable employe est de type Employe
employe.age = 19;
employe.salaire = employe.salaire * 1.1;
```

## 2. Structures

Les structures sont, comme les classes, des ensembles cohérents de données et de fonctionnalités. Elles sont similaires aux classes mais ont comme principale différence d'être des **types valeur**.

Cette différence implique des comportements différents, notamment le fait qu'un objet de type structure ne peut pas être `null` et a donc forcément une valeur. Lors de la déclaration d'une variable de type structure, chacun des membres de l'objet est initialisé à sa valeur par défaut.

On déclare une structure presque de la même manière qu'une classe, mais en utilisant le mot-clé `struct`.

```
<modificateur d'accès> [partial] struct <nom de la structure>
{
}
}
```

Les structures héritent toutes implicitement de la classe `System.ValueType`, elle-même héritant de la classe `System.Object`, mais elles ne peuvent pas hériter d'une autre classe ou structure. Elles ne peuvent pas avoir de destructeur et ne peuvent pas avoir de constructeur sans paramètre. Toutefois, il n'est pas obligatoire de définir un constructeur dans une structure.

```
public struct Voiture
```

```
{  
    public int kilometrage;  
}
```

Pour initialiser notre voiture, on peut écrire :

```
Voiture maFerrari;  
maFerrari.kilometrage = 42;
```

Il est à noter qu'au même titre que les classes, il est possible de définir des structures partielles. Celles-ci sont rarement utilisées car les objets de type structure sont la plupart du temps des conteneurs légers destinés à grouper quelques variables ayant une relation entre elles.

### 3. Création de méthodes

Une méthode est une fonction ou une procédure définie dans une classe ou une structure. Elle définit une fonctionnalité associée à l'objet et peut manipuler les variables de l'objet afin de modifier son état.

#### a. Création

Pour définir une méthode, il suffit d'écrire une procédure ou une fonction à l'intérieur d'une définition de type.

```
public class Voiture  
{  
    private bool estDemarree;  
  
    public void Demarrer()  
    {  
        estDemarree = true;  
        Console.WriteLine("VROUUUUUM");  
    }  
}
```

C# 6 introduit un raccourci syntaxique pour l'écriture de méthodes simples. Il est utilisable lorsqu'une méthode contient une seule instruction, comme un calcul. La méthode est alors définie en lui associant une expression à l'aide de l'opérateur `=>`. La syntaxe utilisée est la suivante :

`<modificateur d'accès> <type> <nom>([paramètres]) => <expression>`

En utilisant cette syntaxe, la procédure `Demarrer` (simplifiée) peut être réécrite ainsi :

```
public void Demarrer() => estDemarree = true;
```

Une fois définie, une méthode peut, comme une variable, être appelée sur un objet grâce à l'opérateur point (`.`), sous réserve que le modificateur d'accès de la méthode l'autorise.

```
maFerrari.Demarrer();
```

Plusieurs méthodes peuvent avoir le même nom, mais elles doivent se distinguer par leurs arguments. On dit alors que ces méthodes sont surchargées.

La distinction entre les différentes surcharges ne se fait pas sur le nom des arguments. Les points importants sont :

- le nombre des arguments,
- le type des arguments,
- l'ordre des arguments.

```
public void EnvoyerCourrier(string nomDestinataire, Adresse
adresseDestinataire)
{
}

public void EnvoyerCourrier(Adresse adresseDestinataire, string
nomDestinataire)
{
}

public void EnvoyerCourrier(string nomDestinataire, Adresse
adresseDestinataire, decimal tarifTimbre)
{
}

public void EnvoyerCourrier(string nomDestinataire, Adresse
adresseDestinataire, string adresseExpéditeur)
{
}
```

Écrire ces quatre méthodes dans la même classe est tout à fait valide, même si la seconde a un intérêt très limité puisqu'elle possède exactement les mêmes paramètres que la première. Chaque appel de méthode sera lié à la bonne déclaration à la compilation, en fonction des paramètres passés dans l'appel.

## b. Méthodes partielles

La version 3 de C# (arrivée avec .NET 3.5) a amené les méthodes partielles pour la même raison que les classes partielles ont fait leur apparition dans C# 2. La génération de code **LINQ to SQL** se fait à l'aide de classes partielles qui permettent au développeur d'ajouter des fonctionnalités au code généré. L'ajout de la notion de méthodes partielles permet en plus d'ajouter des traitements dans le code généré, mais sans le modifier.

Le principe des méthodes partielles est relativement simple : une première partie de la méthode est définie avec le mot-clé `partial` mais elle ne possède pas de bloc de code associé. Ceci permet au compilateur de savoir que la méthode est susceptible d'avoir une implémentation dans une autre partie de la classe.

Les méthodes partielles doivent être définies dans des classes partielles. Dans le cas contraire, le compilateur génère une erreur.

Les méthodes partielles ne peuvent pas être des fonctions. En effet, le code appelant cette fonction dépendrait



d'elles pour fonctionner convenablement, ce qui ne correspond pas à la logique d'extension de fonctionnalité décrite ci-dessus.

Pour la même raison, elles ne peuvent pas avoir de modificateur d'accès, leur visibilité étant directement définie à `private` par le compilateur.

```
partial class Voiture
{
    partial void AuDemarrage();

    void Demarrer()
    {
        AuDemarrage();
    }
}
```

La seconde partie de la méthode peut être définie dans le même fichier ou dans un autre, mais toujours dans la même classe partielle. Elle doit avoir la même signature que la première partie.

```
partial class Voiture
{
    partial void AuDemarrage()
    {
        Console.WriteLine("VROUUUUUM");
    }
}
```

Ainsi, à l'appel de la méthode `Demarrer`, le moteur vrombit !

Si la deuxième partie de cette méthode n'est pas présente, alors la méthode partielle n'a pas d'implémentation. Dans ce cas, le compilateur supprime la définition de la méthode partielle ainsi que les appels à cette méthode.

### c. Méthodes d'extension

Les méthodes d'extension permettent de rajouter des fonctionnalités à des classes ou des structures existantes sans en modifier le code. Ce type de méthode est tout à fait adapté à l'extension de classes du framework .NET telles que la classe `String`. En effet, ces classes ne sont pas modifiables et il arrive fréquemment que l'on souhaite leur adjoindre une fonctionnalité.

Avant C# 3, la fonctionnalité était définie dans une classe utilitaire, éventuellement statique, et appelée de manière conventionnelle.

```
bool estPalindrome = objetUtilitaire.EstUnPalindrome(maChaine);
```

Tout comme les méthodes utilitaires décrites ci-dessus, les méthodes d'extension sont écrites à l'extérieur de la classe étendue mais elles sont utilisables de la même manière que les méthodes de la classe étendue.

Une méthode d'extension doit respecter une convention d'écriture stricte :

- Elle doit être définie dans une classe marquée `static`.
- Elle doit être marquée `static`.
- Son premier paramètre (obligatoire) doit être du type étendu. Il doit être précédé du mot-clé `this`.

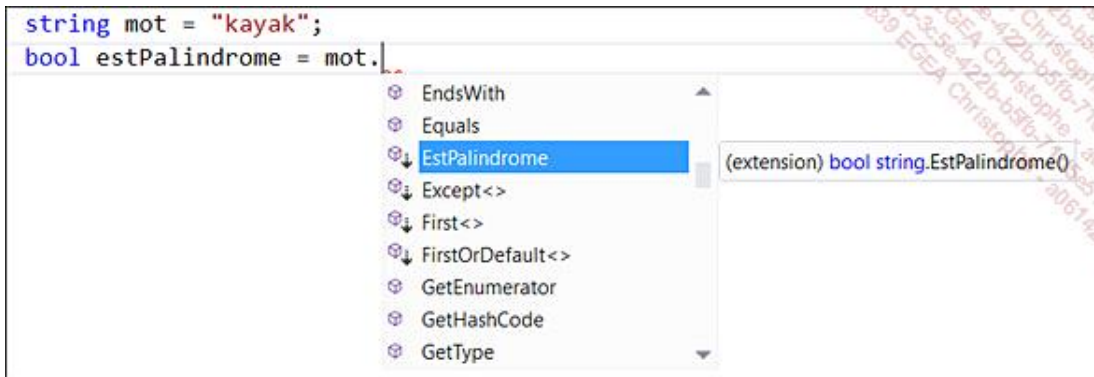
Dans cet exemple, nous ajoutons une méthode à la classe `string` permettant de tester si un mot est un palindrome, c'est-à-dire s'il peut se lire indifféremment de gauche à droite ou de droite à gauche.

```
static class Extensions
{
    public static bool EstPalindrome(this string motATester)
    {
        if (motATester == null)
            return false;

        string motInverse = "";
        for (int i = motATester.Length - 1; i >= 0; i--)
        {
            motInverse = motInverse + motATester [i];
        }

        return motInverse.Equals(motATester);
    }
}
```

Si nous utilisons ensuite une variable de type `string`, la méthode `EstPalindrome` est disponible et suggérée par IntelliSense.



- Les méthodes d'extension sont repérées dans la liste par une icône différente des méthodes classiques ainsi que par la précision "(extension)" dans le texte d'information associé.

#### d. Méthodes opérateurs

Les méthodes opérateurs permettent de redéfinir un opérateur standard du langage pour l'utiliser sur des types définis par le développeur.

Dans le cas de deux collections de timbres, redéfinir l'opérateur `+` permettrait de fusionner les collections.

Prenons deux objets de type `CollectionDeTimbres` définis comme ci-dessous et essayons de les

additionner :

```
struct CollectionDeTimbres
{
    public int nombreDeTimbres;
    public int prixEstime;
}
```

```
CollectionDeTimbres collection1, collection2, collectionResultat;

collection1.nombreDeTimbres = 2500;
collection1.prixEstime = 250;

collection2.nombreDeTimbres = 1240;
collection2.prixEstime = 200;

collectionResultat = collection1 + collection2;

Console.WriteLine("La nouvelle collection contient {0} et son prix est estimé à {1}.",
    collectionResultat.nombreDeTimbres, collectionResultat.prixEstime);
```

Impossible d'appliquer l'opérateur '+' aux opérandes de type 'CollectionDeTimbres' et 'CollectionDeTimbres'

Le compilateur ne sait pas comment additionner ces deux éléments et nous le fait savoir en affichant une erreur bloquante pour la compilation. Rajoutons une redéfinition de l'opérateur + à notre structure :

```
public static CollectionDeTimbres operator +(CollectionDeTimbres
collection1, CollectionDeTimbres collection2)
{
    CollectionDeTimbres resultat;
    resultat.nombreDeTimbres = collection1.nombreDeTimbres +
collection2.nombreDeTimbres;
    resultat.prixEstime = collection1.prixEstime +
collection2.prixEstime;

    return resultat;
}
```

Après cet ajout, l'erreur de compilation disparaît et l'exécution du code affiche le résultat suivant :

```
La nouvelle collection contient 3740 timbres et son prix
est estimé à 450€
```

## 4. Création de propriétés

Les classes peuvent contenir des données publiques sous la forme de variables membres, mais afin de respecter le concept d'encapsulation, il est préférable d'exposer des propriétés plutôt que des variables. En effet, les propriétés permettent d'exécuter du code à chaque accès en lecture ou en écriture. Ceci permet de valider les données et ainsi de maintenir l'objet dans un état cohérent.

### a. Lecture et écriture

Une propriété est semblable à une déclaration de fonction mais elle contient deux blocs de code identifiés par les mots-clés `get` et `set`. Le bloc `get` est exécuté à la lecture de la propriété tandis que le bloc `set` est exécuté lorsque l'on accède en écriture à la propriété.

Une propriété ressemble à ceci :

```
private int kilometrage;

public int Kilometrage
{
    get
    {
        return kilometrage;
    }
    set
    {
        if (value > kilometrage)
        {
            kilometrage = value;
        }
        else
        {
            Console.WriteLine("Le kilométrage d'une voiture ne
peut pas diminuer");
        }
    }
}
```

Afin de respecter le principe d'encapsulation, la variable `kilometrage` est maintenant marquée comme `private`.

On pourra remarquer le mot-clé `value` que l'on trouve dans le bloc `set`. Celui-ci représente la valeur assignée à la propriété. Un test a aussi été ajouté dans le bloc `set` afin de valider la valeur passée à la propriété.

## b. Lecture seule

Il peut être intéressant pour la cohérence d'un objet de n'exposer une propriété qu'en lecture seule. Dans notre cas, on peut dire que le kilométrage d'une voiture ne doit être modifié que par le fonctionnement interne de la voiture. Protéger la variable d'instance `kilometrage` en l'exposant uniquement à travers une propriété en lecture seule paraît donc être une excellente solution.

Pour cela, il suffit de déclarer la propriété sans bloc `set`. Sans ce bloc, toute tentative d'assignation à la propriété résulte en une erreur de compilation. Notre kilométrage est sauf !

```
private int kilometrage;

public int Kilometrage
{
    get
    {
        return kilometrage;
    }
}
```

```
}
}
```

C# 6 introduit un raccourci syntaxique pour l'écriture de ce type de propriété. Lorsque le contenu du bloc `get` est très simple, la propriété peut être définie en lui associant une expression à l'aide de l'opérateur `=>`. La syntaxe utilisée est la suivante :

```
<modificateur d'accès> <type> <nom> => <expression à évaluer>
```

En utilisant cette syntaxe, la propriété `Kilometrage` peut être réécrite ainsi :

```
private int kilometrage;
public int Kilometrage => kilometrage;
```

Une seconde propriété fournissant la valeur du kilométrage suivie du symbole "km" pourrait ressembler au code ci-dessous.

```
public string KilometrageString => string.Format("{0} km", kilometrage);
```

### c. Écriture seule

Dans de très rares cas, il peut être utile de définir une propriété en écriture seule. Par exemple, on peut avoir besoin de définir une propriété `MotDePasse` dans une classe `Utilisateur`. Il n'est clairement pas souhaitable que ce mot de passe puisse être lu. Une propriété en écriture seule semble donc tout indiquée pour l'implémenter.

On implémente une propriété en écriture seule en omettant le bloc `get`.

```
private string motDePasse;

public string MotDePasse
{
    set
    {
        motDePasse = value;
    }
}
```

### d. Propriétés automatiques

Lorsqu'aucune logique particulière ne doit être associée aux accesseurs `get` et `set` d'une propriété, il peut être intéressant d'utiliser les propriétés automatiques, arrivées avec C# 3 et .NET 3.5. On déclare ces éléments en utilisant un raccourci syntaxique pour la définition de propriétés.

Une propriété automatique est déclarée comme une propriété classique, mais en ne fournissant pas de corps aux blocs `get` et `set`. Ajoutons une propriété `Couleur` à notre `Voiture` :

```
public string Couleur { get; set; }
```

Comme cette propriété n'accède pas explicitement à une variable membre, il n'est pas nécessaire d'en déclarer une pour stocker la valeur qui lui est affectée. Dans les coulisses, le compilateur en créera une et générera le corps des blocs `get` et `set`. Ci-dessous, une représentation C# du code IL généré par le compilateur.

```
private string couleur ;
public string Couleur
{
    get { return couleur; }
    set { couleur = value; }
}
```

L'un ou l'autre des blocs `get` et `set` peut être marqué par un modificateur d'accès afin de protéger l'accès à la propriété (n'exposer publiquement que de la lecture seule, par exemple).

```
public string Couleur { get; private set; }
```

Le modificateur utilisé pour un bloc doit être plus restrictif que le modificateur appliqué à la propriété. Spécifier un modificateur d'accès aux deux blocs résulte en une erreur de compilation, même si les deux blocs ont le même modificateur d'accès.

## e. Initialisation de propriétés automatiques

Jusqu'à la cinquième version du langage C#, il n'était pas possible d'initialiser une propriété automatique à la suite de sa déclaration : il était obligatoire de le faire à partir d'un bloc de code, comme une méthode ou le corps d'une autre propriété. Depuis l'arrivée de C# 6, il est tout à fait possible d'effectuer cette action. La portion de code suivante déclare une propriété automatique et lui affecte la valeur "Vert".

```
public string Couleur { get; set; } = "Vert";
```

La valeur utilisée pour l'initialisation peut également être retournée par une expression, comme un appel de fonction ou un calcul.

```
public string Couleur { get; set; } = TrouverNomCouleurRGB(0, 255, 0);
```

## f. Propriétés automatiques en lecture seule

La possibilité d'initialiser une propriété au moment de sa déclaration amène à C# la capacité de créer des propriétés automatiques en lecture seule. Sans initialisation, une propriété automatique en lecture seule n'aurait en effet aucune utilité.

Ce type de propriété est déclaré de la même manière qu'une propriété automatique avec initialisation. La seule différence est la suppression du mot-clé `set` pour obtenir une propriété en lecture seule. La propriété `Couleur` définie plus haut peut être déclarée en lecture seule de la manière suivante :

```
public string Couleur { get; } = "Vert";
```

Ci-dessous, une représentation C# du code IL généré par le compilateur.

```
private readonly string couleur;  
public string Couleur  
{  
    get { return couleur; }  
}
```

Le code IL du constructeur de la classe contenant la propriété se voit également modifié pour inclure une instruction initialisant la variable membre `couleur`.



Le mot-clé `readonly` utilisé pour la déclaration de la variable membre générée indique que celle-ci n'est modifiable que dans le constructeur de la classe qui la contient.

## g. Propriétés indexées

Les propriétés indexées, aussi nommées indexeurs ou propriétés par défaut, permettent un accès de type tableau à un ensemble d'éléments.

Ces propriétés acceptent un ou plusieurs paramètres passés entre crochets. Elles n'ont pas de nom et une classe peut posséder plusieurs propriétés indexées qui doivent se différencier par leurs paramètres.

On déclare une propriété indexée en utilisant la syntaxe suivante :

```
<modificateur d'accès> <type> this[paramètre1, [paramètre2,  
paramètre3...]]  
{  
    //Accesseurs get et/ou set  
}
```

Une implémentation (parfaitement inutile) de propriété indexée pourrait être la suivante :

```
public class Voiture  
{  
    public string this[int numeroRoue]  
    {  
        get { return "Roue n°" + numeroRoue; }  
    }  
}
```

Cette propriété indexée peut ensuite être utilisée de la manière suivante :

```
//voiture est de type Voiture  
Console.WriteLine(voiture[2]);
```

Ce qui affiche : Roue n° 2

## 5. Membres statiques

Les membres statiques (ou membres partagés) d'une classe sont des variables, des propriétés ou des méthodes qui ne sont pas spécifiques à une instance de la classe, mais à la classe elle-même. Un membre statique existe en un seul exemplaire. Accéder à ce membre à travers un objet n'aurait donc pas de sens, c'est pourquoi on l'atteint à travers le nom du type auquel il est associé. Essayer de l'utiliser à travers une instance déclenche une erreur de compilation.



Un membre statique ne peut pas non plus accéder à un membre d'instance de sa classe, sauf si on lui fournit l'objet dont on veut utiliser le membre.

On définit un membre partagé en utilisant le mot-clé `static` après le modificateur d'accès.

Ici, nous considérerons que le nombre de roues est une caractéristique identique pour toutes les voitures. Il est donc possible d'ajouter une propriété statique définissant ce nombre dans la classe `Voiture` :

```
public static int NombreDeRoues { get { return 4; } }
```

Pour utiliser cette propriété, nous écrirons :

```
int nombre = Voiture.NombreDeRoues;
```

Il en va de même pour tous les types de membres qu'une classe ou une structure peut avoir.



Le mot-clé `static` ne peut pas être combiné avec le mot-clé `const`, les constantes étant des types particuliers de membres statiques.

## 6. Utilisation des classes et structures

Le cas général d'utilisation d'une classe ou d'une structure passe par la construction d'un objet. Une fois créé, il est possible de lire ou d'écrire des données dans l'objet, ainsi que d'exécuter les méthodes qui lui sont associées.

### a. Instanciation

L'instanciation d'un objet à partir d'une classe se fait en utilisant le mot-clé `new` :

```
//Voiture est une classe  
  
//Déclaration de la variable. Comme Voiture est une classe,
```



```
//auto vaut null
Voiture auto;

//Instanciation de l'objet auto. Celui-ci n'est plus null
auto = new Voiture();
```

Pour les structures, les choses sont un peu différentes. Comme une structure est un type valeur, une variable de type structure n'est pas null au moment de sa déclaration et chacun de ses membres a pour valeur la valeur par défaut de son type.

```
//Voiture est une structure

//Déclaration de la variable
Voiture auto;
```

Si un constructeur a été défini pour le type structure, alors il est possible d'initialiser notre objet de la même manière que l'on instancie un objet à partir d'une classe.

```
Voiture auto;
//Le constructeur de Voiture a pour paramètre : int kilometrage
auto = new Voiture(42);
```

## b. Initialisation

L'initialisation d'un objet est un processus dont les éléments sont la construction d'une instance et l'affectation de valeurs à ses propriétés. Classiquement, on initialise les propriétés d'un objet de la manière suivante :

```
Voiture auto = new Voiture();
auto.Couleur = "Rouge";
auto.Kilometrage = 12500;
auto.Prix = 20000;
```

Afin de simplifier ce processus qui peut vite devenir fastidieux, C# 3 a introduit les initialiseurs, qui permettent de créer notre objet et d'initialiser tout ou partie de ses propriétés publiques en une seule instruction.

```
Voiture auto = new Voiture { Couleur = "Rouge", Kilometrage =
12500 };
auto.Prix = 20000;
```

## c. Types anonymes

Durant le développement d'une application, il est souvent utile de créer de petits objets qui seront utilisés dans une ou deux fonctions tout au plus. Ces objets ont une durée de vie très courte et n'ont plus ou moins aucune responsabilité vis-à-vis du fonctionnement de l'application.

Créer des classes comme modèles pour ces objets est donc une contrainte n'apportant pas vraiment de valeur à

l'application.

Depuis C# 3, il n'est plus nécessaire de créer des types spécifiques pour ces objets. En effet, les types anonymes ont exactement le rôle décrit ci-dessus.

On peut créer un objet de type anonyme en utilisant presque exactement la même syntaxe que nous avons vue pour les initialiseurs. Il suffit juste de ne pas spécifier de nom de type après le mot-clé `new`, et déclarer chacune de nos propriétés en donnant leur nom et leur valeur.

```
new { Couleur = "Rouge", Kilometrage = 12500 }
```

Par contre, un problème se pose... Comment déclarer la variable qui va contenir cet objet ?

Eh bien, nous sommes typiquement dans le cas où l'**inférence de type** et le mot-clé `var` de C# trouvent toute leur valeur. En effet, le compilateur C# est capable de générer un type d'après les propriétés que nous avons spécifiées pour la création de notre objet. Ce type inconnu durant la phase de développement prendra alors vie durant la compilation et notre variable déclarée avec le mot-clé `var` sera liée au même moment à ce nouveau type.

Pour le vérifier, faisons un essai :



Avant même la compilation, Visual nous montre que notre type est bien reconnu comme anonyme et qu'il possède bien trois propriétés de type `string`. Il possède aussi quatre méthodes que vous aurez peut-être reconnues : ce sont les méthodes fournies par la classe `System.Object`. En effet, comme toutes les classes, notre type anonyme hérite lui aussi de la classe `Object`.

Ces méthodes sont les seules que pourra avoir notre objet, car il est impossible de définir une méthode dans un objet anonyme.

Le résultat de l'exécution de notre code est le suivant :

```
adresse est de type  
<>f__AnonymousType0`3[System.String,System.String,System.String]
```

On peut constater que le compilateur a effectivement généré un type (au nom un peu barbare) et qu'il a bien lié la variable `adresse` à ce type. Par contre, instancier une variable de type anonyme ne veut pas nécessairement dire que le compilateur générera une classe pour cette variable. Il cherchera d'abord un type qu'il a généré qui correspond à la variable que nous souhaitons créer, et en créera un si aucun ne correspond.

Pour comprendre quels critères sont utilisés par le compilateur, créons quelques variables et comparons le résultat avec notre résultat précédent :

```
//2 variables déclarées exactement de la même manière
var adresse = new { Rue = "rue de la gare", Ville = "Lyon", CodePostal = "69001" };
var adresseIdentique = new { Rue = "rue de la gare", Ville = "Lyon", CodePostal = "69001" };
//4 propriétés au lieu de 3
var adresseNombre = new { Numero = 15, Rue = "rue de la gare", Ville = "Lyon", CodePostal = "69001" };
//Une propriété avec un nom différent
var adresseNom = new { Rue = "rue de la gare", Ville = "Lyon", CP = "69001" };
//CodePostal et Ville ont été interchangés
var adresseOrdre = new { Rue = "rue de la gare", CodePostal = "69001", Ville = "Lyon" };
//CodePostal est maintenant un entier
var adresseType = new { Rue = "rue de la gare", Ville = "Lyon", CodePostal = 69001 };

Console.WriteLine($"adresse est de type {adresse.GetType()}");
Console.WriteLine($"adresseIdentique est de type {adresseIdentique.GetType()}");
Console.WriteLine($"adresseNombre est de type {adresseNombre.GetType()}");
Console.WriteLine($"adresseNom est de type {adresseNom.GetType()}");
Console.WriteLine($"adresseOrdre est de type {adresseOrdre.GetType()}");
Console.WriteLine($"adresseType est de type {adresseType.GetType()}");
```

Le résultat est le suivant :

adresse est de type

```
<>f__AnonymousType0`3[System.String,System.String,System.String]
```

adresseIdentique est de type

```
<>f__AnonymousType0`3[System.String,System.String,System.String]
```

adresseNombre est de type

```
<>f__AnonymousType1`4[System.Int32,System.String,System.String,
System.String]
```

adresseNom est de type

```
<>f__AnonymousType2`3[System.String,System.String,System.String]
```

adresseOrdre est de type

```
<>f__AnonymousType3`3[System.String,System.String,System.String]
```

adresseType est de type

```
<>f__AnonymousType0`3[System.String,System.String,System.Int32]
```

Les deux premières variables sont du même type, mais toutes les autres ont un type différent. On en déduit donc que les critères de choix sont les suivants :

- le nom des propriétés,
- le nombre de propriétés,
- l'ordre de définition des propriétés,
- le type des propriétés.

Il est à noter qu'il est possible de comparer deux instances de la même classe anonyme en utilisant la méthode `Equals` (héritée de la classe `Object`). Celle-ci est redéfinie par le compilateur dans le type généré, et permet de dire que les variables `adresse` et `adresseIdentique` sont égales :

```
Console.WriteLine(adresse.Equals(adresseIdentique));
```

La méthode `Equals` renverra `false` si l'une de ces conditions n'est pas respectée :

- Les deux objets comparés doivent être du même type.
- Ils doivent être déclarés dans le même assembly.
- Les valeurs de chaque propriété des objets doivent être identiques.