Encapsulation

Jusqu'ici, nous avons vu les mots-clés ***private, protected*** et ***public***[1] devant les propriétés et méthodes. Mais à quoi servent-ils ?

Ils définissent la visibilité de la propriété ou méthodes d'un objet au sein d'un programme.
private : accessible uniquement depuis l'intérieur de la classe
protected : accessible depuis l'intérieur de la classe et les descendants (héritage)
public : accessible à l'extérieur et à l'intérieur de la classe
Prenons un exemple pour illustrer ce cas. Dans les anciennes voitures, le conducteur a la possibilité d'allumer les phares en tournant un bouton par exemple. La méthode « allumer les phares » doit être accessible pour le conducteur, elle a le mot-clé ***public***.
C# : Le conducteur démarre la voiture et allume les phares, il se trouve en dehors de la classe et appelle une méthode de la classe

```
porsche.start();
porsche.toggleHeadlights(true);
```

Dans une nouvelle voiture, les phares vont s'allumer automatiquement au démarrage. Le conducteur n'a pas besoin d'avoir un bouton, c'est la voiture qui le fait pour lui. La méthode « allumer les phares » a le mot-clé ***private***. Elle est accessible uniquement dans l'objet.

C# : cette fois, c'est au démarrage de la voiture que les phares s'allument automatiquement. Le conducteur ne fait que démarrer la voiture.

```
porsche.start();
```

[1] D'autres mots-clés de visibilité existent dans les langages de programmation (package, internal, ...)

C# : Code de la classe, la méthode start appelle la méthode allumer les phares.

```
class Car
{
    // Properties
    private string _color;
    private int _doorNumber;

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="color">color of car</param>
    /// <param name="doorNumber">number of door</param>
    public Car(string color, int doorNumber)
    {
        _color = color;
        _doorNumber = doorNumber;
    }

    /// <summary>
    /// Start a car
    /// </summary>
    public void start()
    {
        this.toggleHeadlights(true);
        // TODO : create code
    }

    /// <summary>
    /// Stop a car
    /// </summary>
    public void stop()
    {
        toggleHeadlights(false);
        // TODO : create code
    }

    /// <summary>
    /// Turn on or turn off headlights for a car
    /// </summary>
    /// <param name="toogle">Turn on or turn off headlights</param>
    private void toggleHeadlights(bool toogle)
    {
        // TODO : create code
    }
}
```

Définition:

> L'encapsulation est l'ensemble des techniques mises en place dans le but de n'offrir à l'utilisateur d'un objet que les attributs (données) et méthodes (actions) dont il a véritablement besoin, lui fournissant ainsi l'interface la plus simple possible.

Dans l'exemple ci-dessus, on a décidé que les phares d'une voiture devaient s'allumer et s'éteindre de manière automatique au démarrage et à l'arrêt de la voiture. La méthode qui change véritablement l'état des phares n'est donc pas mise à disposition de l'utilisateur.

Getter et Setter

Ce qui est appelé Getter et Setter (« accesseur » et « mutateur » en français) sont des méthodes permettant d'accéder aux attributs d'une classe.

Pour l'instant, notre classe a deux propriétés : _color et _doorNumber. Ces dernières sont de visibilité privée. Il n'est pas possible d'y accéder depuis l'extérieur. Si un jour, nous désirons repeindre notre voiture, il serait bien de pouvoir changer sa couleur.
Définition :

Getter : méthode permettant d'accéder à la valeur d'une propriété privée Setter : méthode permettant d'inscrire la valeur d'une propriété privée

Et concrètement, comment on code ces méthodes ?
L'idée de base est d'avoir un attribut privé et une méthode publique pour y accéder ou le modifier. Ainsi on aurait par exemple :

```
class Car
{
    // Properties
    private string _color;

    /// <summary>
    /// GETTER
    /// Return the current color or 'orange' if not set
    /// </summary>
    public string getColor()
    {
        if (_color==null)
            return "orange";

         return _color;
    }

    /// <summary>
    /// SETTER
    /// Set to the given color if not null
    /// </summary>
    public void setColor(string color)
    {
        if (color!=null)
            _color=color;
    }

}
```

Toutefois, en C#, il existe une construction spécifique présentée ci-après et nommée 'propriété'

```
class Car
{
    // Properties
    private string _color;
    private int _doorNumber;

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="color">color of car</param>
    /// <param name="doorNumber">number of door</param>
    public Car(string color, int doorNumber)
    {
        _color = color;
        _doorNumber = doorNumber;
    }


    /// <summary>
    /// Color property definition
    /// </summary>
    public string Color
    {
        get
        {
            return _color;
        }
        set
        {
            _color = value;
        }
    }

    /// <summary>
    /// DoorNumber property definition
    /// </summary>
    public string DoorNumber
    {
        get
        {
            return _doorNumber;
        }
        set
        {
            _doorNumber = value;
        }
    }
}
```

À noter que dans cette version, les propriétés débutent avec une majuscule ce qui fait partie des recommandations Microsoft et que nous suivons à l'ETML.

Il existe une version plus condensée (les « propriétés automatiques ») qui cache la déclaration de l'attribut. Elle peut donc porter à confusion lorsqu'on débute et pire encore amener des incohérences de visibilité.

C# - version 2

```
class Car
{
    // Getter  - Setter
    public string Color {get;set;}
    public int DoorNumber {get;set;}

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="color">color of car</param>
    /// <param name="doorNumber">number of door</param>
    public Car(string color, int doorNumber)
    {
        this.Color = color;
        this.DoorNumber = doorNumber;
    }

    //Le reste de la classe ne change pas
}
```

Dans cette version, les propriétés sont déclarées public, mais elles sont maintenant des méthodes. Les mots-clés ***get*** et ***set*** permettent de mettre en place le mécanisme.

Vie et mort d'un objet

En résumé, un objet prend vie quand une instance de sa classe est créée. Ce mécanisme prend effet grâce au mot-clé ***new*** qui appelle une méthode spéciale nommée ***constructeur***. Ceci alloue directement de la mémoire RAM pour stocker les valeurs de l'instance de l'objet (pour une classe qui n'aurait qu'un attribut int, cela correspondrait à 32/64 bits selon l'architecture)

Donc, ma voiture est créée, mais maintenant elle est vieille et doit passer à la casse, je fais comment pour la détruire ?

Quand l'objet n'est plus utilisé, la mémoire qu'il utilise pour son stockage ne peut pas servir à autre chose tant que l'objet est toujours présent (référence utilisée dans le programme). Tout comme, il est impossible de parquer une voiture sur une place où une voiture est déjà parquée.

Dans les langages de programmation modernes tels que C#, la destruction d'un objet se fait de manière automatique via le garbage collector ou ramasse-miettes géré par la machine virtuelle du langage (CLR pour C#). L'avantage de détruire un objet qui n'est plus utilisé est de libérer des ressources mémoires. En effet, tout objet créé utilise de la mémoire pour être stocké (tout comme une voiture utilise un espace pour être parquée).