



## Concepteur Développeur en Informatique

### Développer des composants d'interface

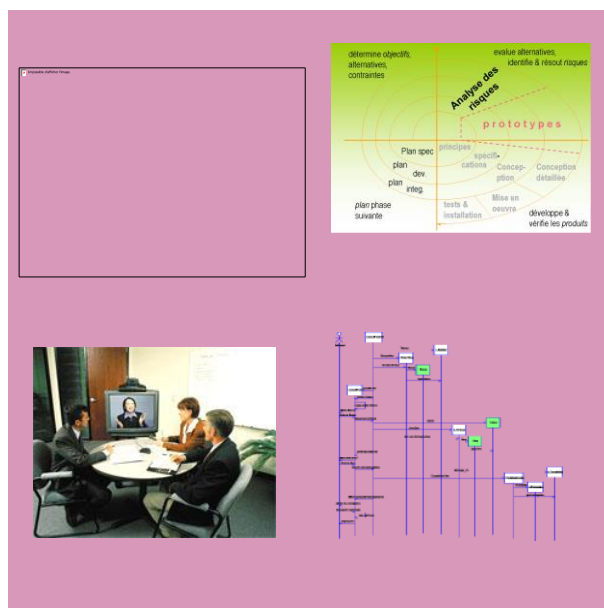
### Programmation Orientée Objet – Partie 1

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S01

## SOMMAIRE

<b>1. Introduction.....</b>	<b>4</b>
1.1. Objectifs .....	4
<b>2. Notion de classe .....</b>	<b>5</b>
2.1. Encapsulation .....	5
<b>3. Définition d'une classe.....</b>	<b>6</b>
3.1. Définition des propriétés d'une classe.....	6
3.2. Accès aux propriétés d'une classe.....	7
3.3. Utilisation de this.....	9
3.4. Contrôle des valeurs d'une propriété d'une classe.....	9
3.5. Propriétés calculées.....	10
3.6. Implémentation des méthodes .....	11
3.6.1. Passage de paramètres aux méthodes .....	11
3.7. Instanciation .....	12
<b>4. Protection et accès aux données membres.....</b>	<b>13</b>
4.1. Restriction des accès.....	13
4.2. Restriction des accès aux classes.....	13
4.3. Restriction des accès aux membres.....	13
<b>5. Construction et destruction des objets .....</b>	<b>14</b>
5.1. Objectifs .....	14
5.2. Constructeurs .....	14
5.3. Surcharge des constructeurs.....	14
5.3.1. Constructeur d'initialisation.....	14
5.3.2. Appel d'un constructeur.....	14
5.3.3. Constructeur par défaut.....	15
5.3.4. Constructeur de copie .....	15
5.4. Destructeur .....	16
5.4.1. Définition d'un destructeur.....	16
<b>6. Déterminer le type d'un objet.....</b>	<b>17</b>
6.1. Méthode GetType et expression typeof.....	17
6.2. L'opérateur is.....	17
6.3. L'opérateur as.....	18
6.4. Traiter les éléments d'un tableau fonction de leur type.....	18
6.5. Conversions boxing - unboxing.....	19
<b>7. Propriétés et méthodes de classes .....</b>	<b>20</b>
7.1. Propriétés de classe.....	21
7.2. Méthodes de classe.....	22

<b>7.1. Classes statiques .....</b>	<b>22</b>
-------------------------------------	-----------

## 1. Introduction

Les prochaines séances d'apprentissage ayant pour thème les objets et la classification doivent vous permettre de vous approprier les bases de la Programmation Orientée Objet. Il ne s'agit ici que d'une initiation à cette technique de programmation et aux concepts qu'elle recouvre.

A l'issue de ces premières séances d'apprentissage, vous devriez être capable de mettre en œuvre et d'intégrer dans vos développements des composants logiciels du Framework .Net proposés sous forme de classes et d'objets.

L'objectif n'est donc pas de maîtriser la conception et la réalisation d'architectures de composants logiciels reposant sur les techniques de Conception Orientée Objet et Programmation Orientée Objet. Ces méthodes et techniques seront plus particulièrement étudiées dans l'unité 3 : conception et réalisation d'applications dans des architectures n-tiers.

Le langage retenu pour illustrer cette technique est C# mais nous aurions pu tout aussi bien choisir un autre langage de la plateforme Dot Net tel que VB.

Chaque exercice est structuré de la façon suivante :

- Description des objectifs visés.
- Explications des techniques à utiliser (Ce qu'il faut savoir).
- Enoncé du problème à résoudre (Travail à réaliser).
- Renvois bibliographiques éventuels dans les ouvrages traitant de ces techniques (Lectures).

Pour ne pas complexifier la mise en œuvre de ces techniques, le champ de l'étude est restreint à la programmation en mode Console.

Pour pouvoir utiliser à bon escient ce support, les premières bases du langage de programmation C# doivent être acquises : manipulation de variables, structure d'un programme C#.

La programmation orientée objet présente une rupture par rapport au développement classique procédural. Elle nécessitera pour les « Cobolistes et assimilés » (et oui il en existe encore) une sérieuse remise en question de leurs habitudes.

Pour les personnes qui intègrent ces approches dès leur initiation à la programmation, la POO constitue un avantage du fait du lien logique qui existe entre les objets naturels et les objets informatiques. Cet avantage reste toutefois à vérifier dans la pratique....

Ce premier document traite des fondamentaux de la programmation objet.

En pratique, les objets seront uniquement manipulés en mémoire et nous ne nous soucierons pas d'assurer leur persistance. Effleuré dans ce premier document, le principe de persistance des objets et de leurs collections sera approfondi par la suite.

### 1.1. Objectifs

Vous allez découvrir les principes de :

- La classification
- L'encapsulation des propriétés et des méthodes.
- L'instanciation.

## 2. Notion de classe

Créer une classe consiste en la définition d'un nouveau **type** de données. Il s'agit de **modéliser** de la manière la plus juste un objet, à partir des possibilités offertes par un langage de programmation.

Il faut donc énumérer toutes les **propriétés** de cet objet et toutes les **opérations** qui vont permettre de définir son comportement.

Les **opérations**, désignées le plus souvent comme fonctions ou méthodes, peuvent être classées de la façon suivante :

- Les opérations de calcul. S'il est nécessaire de réaliser des calculs sur ces données, des fonctions de calcul devront être implémentées.
- Les opérateurs relationnels. Il faut au moins pouvoir tester si deux données sont égales. S'il existe une relation d'ordre pour le nouveau type de donnée, il faut pouvoir aussi tester si une donnée est inférieure ou supérieure à une autre.
- Les opérations de conversion. Si des conversions vers d'autres types de données sont nécessaires, il faudra implémenter les fonctions correspondantes.
- Les opérations de contrôle de l'intégrité de l'objet. La manière dont est modélisé le nouveau type de données n'est probablement pas suffisant pour représenter l'objet de façon exacte. Par exemple, si l'on représente une fraction par un couple de deux entiers, il faudra vérifier que le dénominateur d'une fraction n'est pas nul. Comme autres exemples nous pourrions citer la vérification de la conformité d'une adresse mail, d'un code postal, d'un numéro de téléphone...
- Les opérations de persistance. Ces fonctions, qui s'appuient sur des services de Gestion de la Persistance, seront abordées par la suite. Ces mécanismes sont désignés sous le vocable de la **sérialisation**. Elles permettent de conserver l'état des objets dans une mémoire non volatile ?

La déclaration d'un nouveau type de données et les fonctions qui permettent de gérer les objets associés constituent la **classe** de l'objet.

Les propriétés de l'objet seront implantées sous la forme de **données membres** de la classe. Les différentes fonctions ou méthodes seront implémentées sous la forme de **fonctions membres** de la classe.

De façon courante, dans le *patois* de la programmation orientée objet, **données membres** et **fonctions membres** de la classe sont considérées respectivement comme synonymes de **propriétés** et **méthodes** de l'objet.

### 2.1. Encapsulation

**L'encapsulation** est un concept important de la définition des classes et donc de la Programmation Orientée Objet.

**L'encapsulation** permet de rassembler les **propriétés** composant un objet et les **méthodes** pour les manipuler dans une seule entité appelée **classe** de l'objet.

L'encapsulation n'autorise pas l'accès aux éléments internes à l'objet qui peuvent constituer une grande complexité. L'encapsulation ne donne accès qu'aux éléments de la classe exposés au travers des interfaces de celle-ci.

**L'encapsulation permet ainsi de protéger l'intégrité de l'objet et de nous masquer toute la complexité des calculs effectués en interne.**

L'idée est que le programme qui utilise une classe n'a pas à se soucier du fonctionnement interne de celle-ci.

Par exemple, lorsque vous utilisez la méthode `WriteLine()` de la classe `Console`, vous n'avez pas à vous soucier des mécanismes internes mis en œuvre et de la façon dont cette méthode organise physiquement les données à afficher.

On peut donc considérer que l'encapsulation masque certaines informations et dissimule le contenu interne des objets.

Les propriétés et les méthodes de la classe seront déclarées et implémentées dans le bloc défini délimité par **class { }**. Cela constitue l'implémentation du concept d'encapsulation en C#.

### 3. Définition d'une classe

Une classe, en C# se déclare par le mot clé **class** suivi d'un identificateur de classe qui devra respecter les règles de dénomination définie par la norme définie dans le document « Convention de nommage C# - Microsoft » le PascalCase si accessibilité publique, ou camelCase si accessibilité privée.

```
namespace TestClasses
{
    public class MaClasse
    {
    }
}
```

#### 3.1. Définition des propriétés d'une classe

Les propriétés d'une classe sont déclarées, comme des variables, à l'intérieur du bloc de la classe.

Les identifiants de propriété sont par convention définis selon les règles PascalCase. Chaque déclaration de propriété est construite sur le modèle suivant :

```
public TypedelaPropriété NomDeLaPropriété { Bloc }
```

```
public DateTime DateNaissance
{
    // Définition des accesseurs
}
```

Les propriétés peuvent être déclarées à tout moment à l'intérieur du corps de la classe.

Pour une classe **Stagiaire** représentant les objets de type `Stagiaire`, le nom ne fera pas référence à la classe. Ainsi, pour respecter les conventions de nommage, la propriété `Nom` du stagiaire sera référencée sous le terme `Nom` et pas `NomStagiaire`.

### 3.2. Accès aux propriétés d'une classe

Bien qu'il soit possible en C# d'accéder directement aux propriétés, cette approche n'est pas conforme aux standards de l'encapsulation : Ainsi, les propriétés seront accessibles via des accesseurs, de deux types :

- L'accesseur **Get** qui permet de lire la valeur de la propriété
- L'accesseur **Set** qui permet d'en modifier la valeur.

```
public DateTime DateNaissance
{
    get { return (this._dateNaissance); }
    set { this._dateNaissance = value; }
}
```

Ces 2 accesseurs ne sont pas nécessairement présents. Si la donnée est en lecture seule ou en écriture seule (plus rare), il n'y aura qu'un des 2 accesseurs.

L'accès aux propriétés d'une classe se fait par l'opérateur « . ».

Si la propriété nécessite le stockage d'une valeur particulière on lui associera une variable privée, appelée **champ**, qui sera modifiée par le biais de l'accesseur Set dans le cas de la modification de la propriété et retournée par le biais de l'accesseur Get dans le cas de l'extraction de la valeur de la propriété.

```
private DateTime _dateNaissance;
```

Par respect des conventions, les champs (**fields**) d'une propriété sont déclarés comme variables privés, respectent les règles de nommage CamelCase et sont préfixés par le caractère \_ (underscore).

```
private int _idStagiaire = 0;
private string _nom = string.Empty;
private string _prenom = string.Empty;
private DateTime _dateNaissance;
```

**A noter :** Le terme **value** fait toujours référence à la valeur assignée à la propriété.

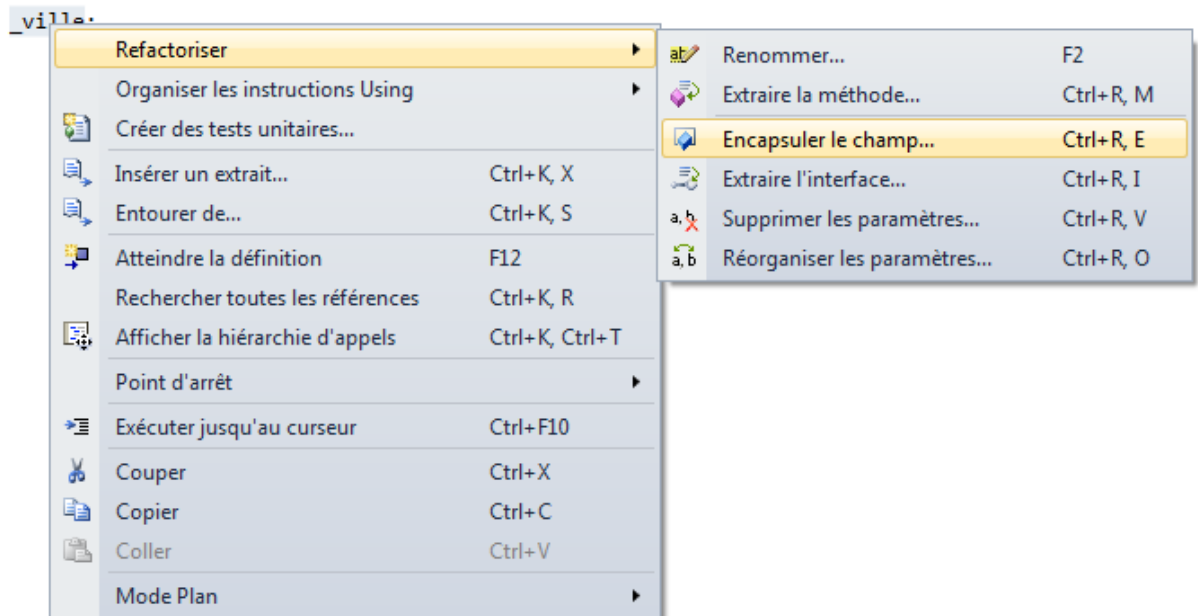
Visual studio offre un assistant qui vous permet d'encapsuler automatiquement un champ.

### Etape 1 : définition du champ

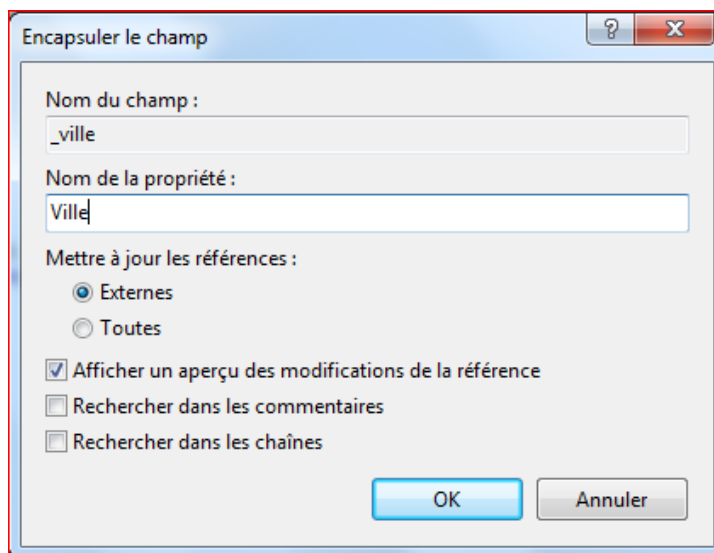
```
private string _ville;
```

### Etape 2 : Encapsulation

Sélectionner le champ puis depuis le menu contextuel obtenu par click droit choisir **Refactoriser** puis **Encapsuler le champ**



Vous obtenez alors la fenêtre de dialogue suivante :



```
private string _ville;

public string Ville
{
    get { return _ville; }
    set { _ville = value; }
}
```

Cliquez sur OK et observez le résultat obtenu. Vous disposez maintenant d'une propriété Ville liée à un champ privé \_ville.



### 3.3. Utilisation de this

Le mot clé **this** fait référence à l'objet courant (i.e. à partir duquel il est appelé).  
Les programmeurs C et C++ connaissent bien cette technique.  
Nous la retrouverons aussi en javascript par exemple.

### 3.4. Contrôle des valeurs d'une propriété d'une classe

Certaines valeurs de propriété devront obéir à des règles précises.  
Il est nécessaire, pour assurer la cohérence de votre système d'information, de vérifier que ces règles aient été respectées.

Dans notre exemple, l'objet **Stagiaire** possède une propriété **Prenom**. Le prénom d'un stagiaire doit obéir aux règles suivantes :

- Sa longueur minimum est de 3 caractères
- Sa longueur ne peut excéder 55 caractères
- Le premier caractère du prénom sera exprimé en majuscule et les caractères suivants en minuscules : *il s'agit ici d'application d'un style et non d'une règle stricte : on pourrait l'envisager sous d'autres approches.*

Au départ, notre propriété se présente ainsi :

```
//// <summary>
/// Prenom du stagiaire
/// </summary>
public string Prenom
{
    get { return (this._prenom); }
    set
    {
        this._prenom = value;
    }
}
```

**Attention :** l'accesseur **get** doit retourner le **champ privé** et non la propriété publique : `return (this.Prenom)` provoquera une exception de type `StackOverflowException` provoquée par une boucle récurrente.  
Mais le code se compilerait très bien !

Ajoutons les règles de vérification de la longueur de la propriété **Nom** afin de nous assurer du respect de l'**intégrité** de nos objets.

Si le programme qui manipule l'**objet stagiaire** cherche à introduire une valeur de **propriété Prenom** non valide, le concepteur de la **classe Stagiaire** doit générer une exception.

Dans un premier temps, nous baserons notre exception sur le modèle de base proposé par le système. Nous verrons par la suite comment implémenter nos propres classes d'exceptions.

**A noter :** La levée d'exception consomme beaucoup de ressources système. Il est donc conseillé dans le programme qui manipule les objets, interface utilisateur par exemple, d'implémenter un contrôle avant d'assigner une valeur à la propriété et donc d'éviter ainsi la levée d'une exception.  
Complément de définition de la propriété **Prenom**.

```
/// <summary>
/// Prenom du stagiaire
/// </summary>
public string Prenom
{
    get { return (this._prenom); }
    set
    {
        if (!isNomPrenomValide(value))
            throw new Exception(string.Format("Le prénom {0} n'est pas valide.", value));
        else
            this._prenom = string.Format("{0}{1}", value.Trim().Substring(0, 1).ToUpper(),
            value.Trim().Substring(1, value.Trim().Length - 1).ToLower());
    }
}
```

Et la méthode interne de contrôle de la chaîne.

```
private bool isNomPrenomValide(string valeur)
{
    if (valeur == null || valeur.Trim().Length < 3 || valeur.Length > 54)
        return false;
    foreach (Char c in valeur)
    {
        if (!char.IsLetter(c) & !char.IsWhiteSpace(c) & c != '-')
            return false;
    }
    return true;
}
```

**A noter** : Utilisation de l'opérateur || plutôt que | Dans cet exemple il ne faut pas évaluer les autres expressions conditionnelles si la première est vraie : si **value** est null, on ne peut manipuler la référence.

### 3.5. Propriétés calculées

Certaines propriétés ne nécessitent pas de stockage et résultent de calculs internes. Elles sont donc associées à des méthodes privées de la classe et ne nécessitent pas de champ associé pour le stockage de la valeur.

C'est l'exemple de l'âge du stagiaire dans le cas de notre classe Stagiaire. La propriété Age, en lecture seule, n'aura donc qu'un accesseur Get. Pour simplification, l'âge est ici calculé de manière approximative...

```
public int Age
{
    get { return calculerAge(); }
}

private int calculerAge()
{
    TimeSpan intervalleTemps;
    intervalleTemps = DateTime.Now - this.DateNaissance;
    return (intervalleTemps.Days / 365);
}
```

### 3.6. Implémentation des méthodes

Les méthodes sont implémentées sous forme de fonctions et obéissent donc aux règles de ces dernières.

Les méthodes d'une classe sont implémentées à l'intérieur d'un bloc {}

Lorsque l'on considère une méthode par rapport à l'objet sur lequel elle s'applique, il faut voir l'objet comme étant sollicité de l'extérieur par un **message**.

Ce **message** peut comporter des paramètres. L'objet doit alors réagir à ce **message** en exécutant cette méthode.

Règles de base d'une méthode :

- Les identifiants de méthodes obéissent aux règles de nommage du PascalCase.
- Si aucun paramètre n'est désigné explicitement entre les parenthèses, le compilateur considère la méthode comme étant sans paramètre. Les parenthèses sont néanmoins obligatoires lors de l'invocation de la méthode.
- Pour renvoyer le résultat de l'exécution des opérations d'une méthode, vous devez utiliser le mot clé **return**.

Méthode qui retourne les valeurs de propriétés d'un objet sous forme d'une chaîne.

```
/// <summary>
/// Chaîne représentant l'objet instancié
/// </summary>
/// <returns>Valeurs des propriétés de l'objet</returns>
public override string ToString()
{
    return string.Format(@"{0};{1};{2};{3};{4};{5};{6};{7};{8};{9};{10}", this._idStagiaire,
        this._nom, this._prenom, this._adresse, this._codePostal, this._ville,
        this._region, this._telephoneFixe, this._telephonePortable, this._adresseMail, this._dateNaissance);
}
```

#### 3.6.1. Passage de paramètres aux méthodes

Si un paramètre est déclaré pour une méthode sans **ref** ni **out**, le paramètre peut avoir une valeur associée. Cette valeur peut être modifiée dans la méthode, mais la valeur de remplacement n'est pas conservée lorsque la procédure appelante récupère le contrôle.

Vous pouvez modifier ce comportement à l'aide d'un mot clé de paramètre de méthode.

**ref** : Le mot clé **ref** fait en sorte que les arguments soient passés par référence. La conséquence est que toute modification apportée au paramètre dans la méthode est reflétée dans cette variable lorsque la méthode appelante récupère le contrôle. Pour utiliser un paramètre **ref**, la définition de méthode et la méthode d'appel doivent toutes les deux **utiliser** le mot clé **ref** explicitement.

**Ref** concerne des arguments en Entrée/Sortie.

**out** : Le mot clé **out** fait en sorte que les arguments soient passés par référence comme pour l'usage de **ref**. Toutefois, la valeur sera déterminée par la méthode. Elle est en Sortie. Pour utiliser un paramètre **out**, la définition de méthode et la méthode d'appel doivent toutes les deux utiliser le mot clé **out** explicitement.

L'exemple ci-dessous illustre ce mécanisme implémenté au niveau de la méthode TryParse de type booléen.

Si la conversion de la chaîne en entier est valide, nombrePostes contiendra la valeur convertie et la méthode retournera true.

```
static int DiagNombresPostes()
{
    int nombrePostes;
    do
    {
        Console.WriteLine("Programme de tris de valeurs");
        Console.WriteLine("Entrez le nombre de valeurs souhaité :");
    } while (!int.TryParse(Console.ReadLine(), out nombrePostes));
    return nombrePostes;
}
```

**params** : Le mot clé **params** vous permet de spécifier un paramètre de méthode avec un nombre d'arguments variable, comme un tableau par exemple.

Aucun paramètre supplémentaire n'est autorisé après le mot clé **params** dans une déclaration de méthode et **un seul mot clé params** est autorisé dans une telle déclaration.

```
public static void MaMethode(params object[] arguments)
{
    |
}
```

### 3.7. Instanciation

Pour qu'un objet ait une existence, il faut qu'il soit **instancié**. Une même classe peut être instanciée **plusieurs fois**, chaque instance ayant des propriétés porteuses de valeurs spécifiques.

En VB ou C#, il n'existe qu'une seule manière de créer une **instance** d'une classe. Cette création d'instance peut se faire en deux temps :

- Déclaration d'une variable du type de la classe de l'objet,
- Instanciation de cette variable par exécution de l'instruction `new`.

Le système génère toujours un constructeur par défaut sans paramètre. Un **constructeur** est une **méthode particulière qui porte le nom de la classe**. Vous pouvez définir plusieurs constructeurs. Nous reviendrons sur ces mécanismes dans le chapitre 6.

Ils se déclarent de la manière suivante en C# :

```
public Stagiaire(int IdStagiaire, string Nom, string Prenom)
{
    this.Nom = Nom;
    this.Prenom = Prenom;
    this.IdStagiaire = IdStagiaire;
}
```

## 4. Protection et accès aux données membres

### 4.1. Restriction des accès

En Programmation Orientée Objet, nous avons vu précédemment que nous ne devons pas accéder directement aux champs d'une classe mais passer par des méthodes spécialisées dites accesseurs afin de respecter le concept d'encapsulation.

Il faut aussi préciser explicitement les conditions de visibilité (et donc d'accès) pour chacune des propriétés et méthodes.

Les restrictions se font par le biais de modificateurs d'accès qui sont au nombre de quatre : `public`, `private`, `protected`, `internal`

### 4.2. Restriction des accès aux classes

Les classes qui ne sont pas imbriquées dans d'autres classes peuvent être publiques ou internes.

Un type déclaré comme public, mot clé **public** est accessible par tout autre type.

Un type déclaré comme interne, mot clé **internal** est uniquement accessible aux types du même assembly.

Par défaut, les classes sont déclarées comme internes.

Les définitions de classes peuvent ajouter le mot clé `internal` pour rendre leur niveau d'accès explicite. Les modificateurs d'accès n'affectent pas la classe elle-même ; elle a toujours accès à elle-même et à tous ses propres membres.

### 4.3. Restriction des accès aux membres

Les membres des classes peuvent être déclarés publics ou internes comme les classes elles-mêmes. Mais ils peuvent être aussi déclarés internes, protégés, ou internes protégés...

Lorsqu'un membre de classe est déclaré comme protégé à l'aide du mot clé **protected**, seuls les types dérivés qui utilisent la classe comme une base peuvent accéder au membre. Nous verrons les types dérivés lorsque nous aborderons l'héritage.

En combinant les mots clés **protected** et **internal**, un membre de classe peut être marqué comme **protected internal** ; seuls les types dérivés ou les types du même assembly peuvent accéder à ce membre.

Enfin, un membre de classe peut être déclaré comme privé avec le mot clé **private**, ce qui indique que seule la classe qui déclare le membre peut accéder à ce membre.

Afin d'implémenter **correctement** le concept d'encapsulation, il convient donc de **verrouiller correctement** l'accès aux champs et de les déclarer **private** ou **protected** tout en permettant leur accès via les méthodes de type **Accesseur** en déclarant ces dernières publiques.

```
private string _nom;  
private DateTime _dateNaissance;
```

Par convention, la **dénomination d'un champ** doit respecter les règles suivantes ;

- commence par le préfixe `_`
- respecte la convention camelCase

## 5. Construction et destruction des objets

### 5.1. Objectifs

- Constructeurs et destructeur des objets
- Propriétés et méthodes de classe

### 5.2. Constructeurs

Quand une instance d'une classe d'objet est créée au moment de l'instanciation d'une variable avec **new**, une fonction particulière est exécutée. Cette fonction s'appelle le **constructeur**. Elle permet, entre autres choses, d'initialiser chaque instance pour que ses propriétés aient un contenu cohérent.

Un constructeur est déclaré comme les autres fonctions membres si ce n'est que l'**identificateur du constructeur** est celui de la classe.

### 5.3. Surcharge des constructeurs

Il peut y avoir plusieurs constructeurs pour une même classe, chacun d'eux correspondant à une initialisation particulière. Tous les constructeurs ont le même nom mais se distinguent par le nombre et/ou le type des paramètres passés (cette propriété s'appelle **surcharge** en programmation objet).

La surcharge des constructeurs nécessite pour être valide que chacun des constructeurs définisse un nombre d'arguments différent ou que les types des arguments déclarés varient. Il s'agit d'une règle commune à la surcharge des méthodes d'une manière générale. On parle de **signature** de méthode. Chaque méthode surchargée doit avoir une signature différente.

Quand on crée une nouvelle classe, il est indispensable de prévoir tous les constructeurs nécessaires. Il est courant de les distinguer en fonction de leur objectif spécifique.

#### 5.3.1. Constructeur d'initialisation

Ce constructeur permet de procéder à une instanciation en initialisant les propriétés, les valeurs de celles-ci étant passées dans les paramètres ou définies sous forme de constante dans la classe.

Il peut exister plusieurs constructeurs d'initialisation avec plus ou moins d'arguments.

```
public Stagiaire(int IdStagiaire, string Nom, string Prenom)
{
    this.Nom = Nom;
    this.Prenom = Prenom;
    this.IdStagiaire = IdStagiaire;
}
```

#### 5.3.2. Appel d'un constructeur

Comme toujours, nous devons **centraliser le coder et éviter de répéter** des opérations afin de faciliter le codage et la maintenance de nos applications. Ainsi nous pouvons appeler un constructeur disposant ou non de paramètre à partir d'un autre constructeur.

Les constructeurs s'appellent en cascade en recourant à la méthode `this()` avec ou sans paramètres.

```
public ClasseA()
{
    // Opérations réalisées pour toute instanciation
}
public ClasseA(string proprieteA)
: this()
{
    this.ProprieteA = proprieteA;
}
public ClasseA(string proprieteA, string proprieteB)
: this(proprieteA)
{
    this.ProprieteA = proprieteA;
}
```

### 5.3.3. Constructeur par défaut

Un constructeur est défini par défaut par le système pour chaque classe si aucun autre constructeur n'est déclaré mais celui-ci ne peut être invoqué que s'il n'existe aucun autre constructeur. Un constructeur par défaut n'accepte aucun paramètre. Il peut être redéfini explicitement comme dans l'exemple ci-dessous :

```
public Stagiaire()
{
}
```

### 5.3.4. Constructeur de recopie

Le constructeur de recopie permet de recopier les propriétés d'un objet existant vers une nouvelle instance de même type. Il prend en paramètre l'objet dont les propriétés doivent être copiées. Les exemples suivants sont des extraits incomplets...

```
// Constructeur de recopie
public Stagiaire(Stagiaire stagiaire)
{
    this._nom = stagiaire.Nom;
    this._dateNaissance = stagiaire.DateNaissance;
}

// Constructeur de recopie
public Stagiaire(Stagiaire stagiaire)
{
    this.Nom = stagiaire.Nom;
    this.DateNaissance = stagiaire.DateNaissance;
}
}
```

**A noter :** Attention à l'utilisation de la première forme ci-dessus qui manipule les **champs privés au lieu des propriétés** : vous n'invoquez par les accesseurs `set` et les contrôles implémentés dans ces derniers ne seront alors pas réalisés.



Nous pouvons aussi utiliser aujourd'hui la technique des initialiseurs, auparavant réservée aux tableaux, avec tous les objets.

La mise en œuvre, légèrement différente, ne pose pas de difficultés particulières. Lors de la création de l'objet, nous allons assigner certaines propriétés de ce dernier sous la forme {propriété1 = valeur1, propriété2 = valeur2}.

```
Salarie salarie = new Salarie() { Nom = "Bost", Prenom = "Vincent", SalaireBrut = 3600 };
```

### 5.4. Destructeur

Dans l'architecture logicielle Dot Net, il n'est pas nécessaire de gérer la mémoire occupée par les objets et nous ne devons pas déclencher explicitement la destruction physique de ceux-ci.

Un instance d'un objet est détruite lorsque les deux conditions suivantes seront réunies :

- Instance inaccessible du fait de la perte de la référence (variable stockant la référence est « morte »)
- Le programme a besoin de ressources mémoire.

Le programme qui se charge de cette tâche s'appelle le **Garbage Collector** ou, en français, le **ramasse-miettes**.

Le **Garbage Collector** est un système capable de surveiller les objets créés par une application, de déterminer quand ces objets ne sont plus utiles, d'informer ces objets et de détruire ces objets pour récupérer leurs ressources. A priori, c'est le système qui décide, en fonction des ressources dont il a besoin, quand faire appel au Garbage Collector.

Dans le cas de notre compteur de classes, si nous souhaitons que le compteur d'instances soit à jour et reflète correctement le nombre d'instances d'objets en mémoire, il est nécessaire de connaître le moment où ces instances seront détruites pour décrémenter la variable **\_compteurInstances**.

C'est pour cela que le **ramasse –miettes** envoie un message à chaque instance avant qu'elle soit détruite, et ceci quelle que soit la classe. Si nous indiquons que le système informe systématiquement des destructions d'objets, c'est que cette méthode doit être implémentée pour tous les objets ...

En fait, toutes les classes que nous créons ou qui sont mises à notre disposition au niveau du Framework, héritent directement ou indirectement de la classe **Object**. Comme vous l'avez vu dans le précédent support pour **string** par exemple, **object** est un alias de la classe **Object** dans .NET Framework. Dans le système de type unifié de C#, tous les types, prédéfinis et définis par l'utilisateur, héritent de la classe de base **Object**. Nous verrons le principe de l'héritage dans l'étape suivante des apprentissages.

#### 5.4.1. Définition d'un destructeur

Un destructeur est une méthode particulière, qui est le pendant du constructeur.

Un destructeur est unique au niveau de la classe et se nomme (tild)~**NomClasse**.



## Programmation Orientée Objet

Dans notre exemple, nous aurons donc un destructeur défini ainsi en charge de décrémenter le compteur d'instances.

```
/// <summary>
/// Destructeur d'instances / mise à jour compteur d'instances
/// </summary>
~Stagiaire()
{
    _compteurInstances--;
}
```

Le compilateur traduira cette méthode en une surcharge de la méthode **Object.Finalize()** qui sera la méthode de réponse à la destruction d'un objet.

Nous n'entrerons pour l'instant pas dans le détail de ces mécanismes.

Des restrictions particulières s'appliquent aux destructeurs. Ils ne peuvent être déclarés avec un modificateur d'accès (public, protected, ..) et sont toujours définis sans paramètres.

## 6. Déterminer le type d'un objet

Nous avons pour satisfaire cet objectif plusieurs mécanismes à notre disposition. Nous mobiliserons ensemble ou alternativement :

- L'opérateur **Is**
- L'opérateur **As**
- La méthode **GetType** disponible sur chaque objet
- L'expression **typeof**

### 6.1. Méthode GetType et expression typeof

La méthode **GetType()** permet d'extraire le type d'un objet et comparer celui-ci au type passé à l'expression **typeof(Type)** qui permet d'obtenir l'objet **System.Type** d'un type et donc de comparer ces deux valeurs.

```
Type type = stagiaire.GetType();
|
```

L'expression suivante pourrait alors être :

```
if (type == typeof(Stagiaire))
```

### 6.2. L'opérateur is

Une expression **is** prend la valeur **true** si l'expression fournie n'est pas **null**, et que l'objet fourni peut être converti en type fourni sans entraîner la levée d'une exception.

Dans l'exemple qui suit, si l'objet est de type **Salarie** alors nous modifions sa propriété **Nom**.

```
if (!(salarie is Salarie)) return false; else ((Salarie)salarie).Nom = "Bost";
```

**Notez** que l'opérateur **is** considère uniquement les conversions de référence, les conversions boxing et les conversions unboxing (voir point suivant). D'autres conversions, comme les conversions définies par l'utilisateur, ne sont pas prises en compte.

### 6.3. L'opérateur as

Assez proche de l'opérateur **is**, **as** peut être plus performant. Il fournit l'objet converti dans son type si c'est possible, sinon une absence de référence. Il y a une seule conversion ici alors qu'il y en a deux dans l'exemple précédent avec **is**.

```
Salarie salarieConverti = salarie as Salarie;  
if (salarieConverti == null) return false;  
else salarieConverti.Nom = "Bost";
```

### 6.4. Traiter les éléments d'un tableau fonction de leur type

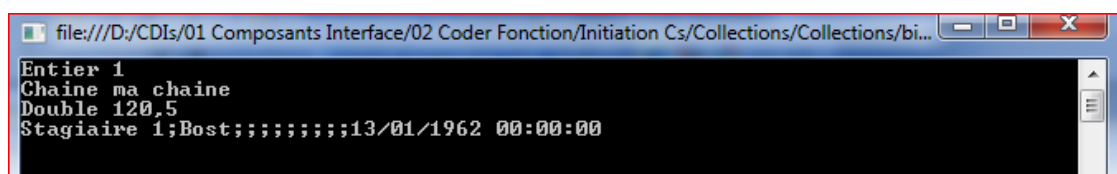
Regardons dans l'exemple quels sont les types stockés dans notre tableau.

Réalisons une liste énumérée des objets stockés dans notre tableau.

```
foreach (object element in Objets)  
{  
    if (element.GetType() == typeof(int))  
    {  
        Console.WriteLine("Entier {0}", element.ToString());  
    }  
    if (element.GetType() == typeof(string))  
    {  
        Console.WriteLine("Chaine {0}", element.ToString());  
    }  
    if (element is double)  
    {  
        Console.WriteLine("Double {0}", element.ToString());  
    }  
    if (element.GetType() == typeof(Stagiaire))  
    {  
        Console.WriteLine("Stagiaire {0}", element.ToString());  
    }  
}
```

Si nous devons faire appel à une méthode spécifique de notre objet, il nous faudra alors le convertir, comme dans les deux exemples précédents.

Vous remarquerez l'invocation de la méthode substituée ToString() du type Stagiaire :

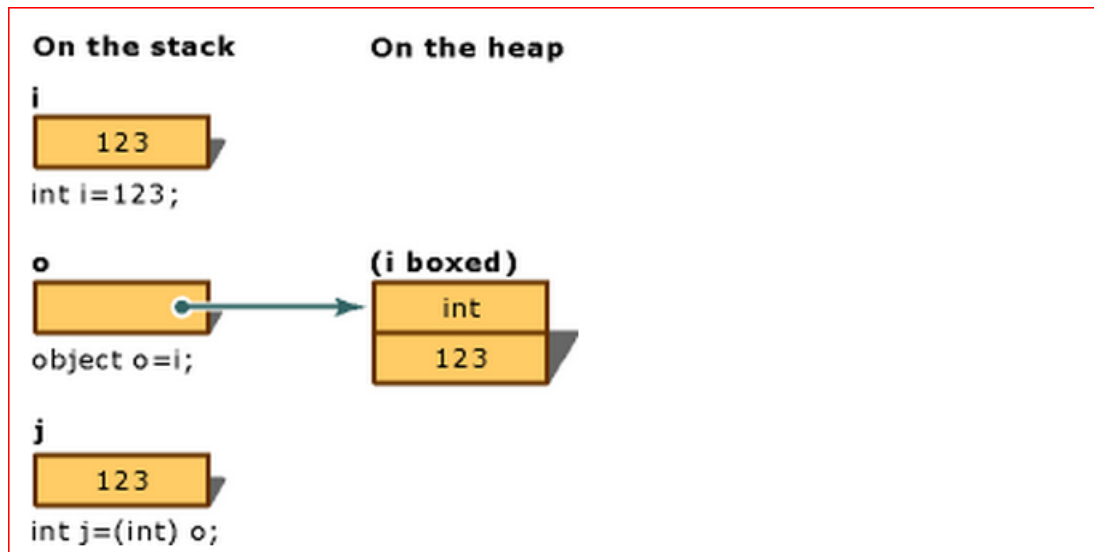


## 6.5. Conversions boxing - unboxing

Comme vous venez de le constater, nous avons pu stocker un entier ou un double dans une collection manipulant des objets.

Hors les objets sont des types gérés par référence et les entiers ou doubles sont des types valeur. Les types valeurs (primitifs) sont gérés sur la pile (stack) et les objets sur le tas (heap) qui permet le stockage en mémoire de grands volumes.

Cette mécanique repose sur la mise en œuvre de conversions de types valeur vers référence et référence vers valeur que vous trouverez dans la littérature désignées comme Conversion boxing et Conversion unboxing. Nous allons nous appuyer sur un premier schéma qui présente ces opérations de conversion :



Puis sur un exemple programmé en C# pour comprendre.

Soient deux variables une de type entier *i* et une variable *o* déclarée comme objet :

```
int i = 55;
object o = i;
Console.WriteLine("valeur de o {0}", o);
Console.WriteLine("valeur de i {0}", i);
```

Nous obtenons sans surprise le résultat suivant :

```
file:///D:/CDIs/01 Composants Interface/02 Codier Fonction/Initiation Cs/Collections/Collections/bi...
valeur de o 55
valeur de i 55
```

Lorsque l'on assigne *i* à l'objet *o*, le runtime (CLR) va attribuer automatiquement un bloc de mémoire sur le tas, copier la valeur de l'entier *i* à ce bloc mémoire puis référencer l'objet *o* à cette copie. Affecté *o* ne modifie pas *i* :

```
o = 56;
Console.WriteLine("valeur de o {0}", o);
Console.WriteLine("valeur de i {0}", i);
```

```
file:///D:/CDIs/01 Composants Interface/02 Codier Fonction/Initiation Cs/Collections/Collections/bi...
valeur de o 56
valeur de i 55
```

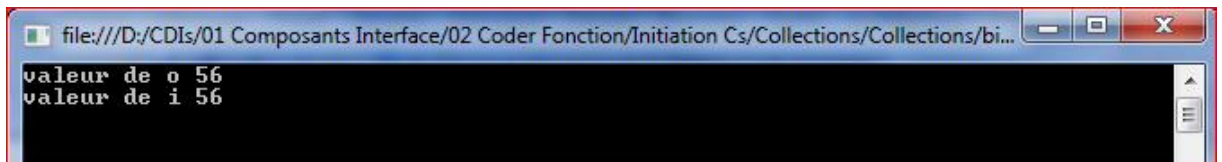
Peut-on alors écrire ceci si nous souhaitons que `i` prenne la valeur de `o` : `i = o;`

Le compilateur refuse de valider cette instruction et nous transmet un message d'erreur qui nous permet de mieux cerner les problèmes de conversion.

Nous devons donc, si nous souhaitons que `i` prenne la valeur de `o`, faire une opération de conversion unboxing, c'est-à-dire convertir (cast) l'objet en `int`.

```
int i = 55;|
object o = i;
o = 56;
i = (int)o;
Console.WriteLine("valeur de o {0}", o);
Console.WriteLine("valeur de i {0}", i);
```

Nous obtenons maintenant le résultat escompté :



Le runtime a fait un gros travail : il a vérifié ce qui était stocké dans l'objet `o` et a transformé son type en `int` : lorsque vous préfixez une variable d'un type entre parenthèses (Type), vous indiquez que vous souhaitez la convertir dans ce type.

`(int)` exprime un cast : convert as type `int`.

Nous aurons recours fréquemment à ces techniques de conversion de type. En effet, le runtime admet peu de conversions implicites pour des raisons de sécurité et de stabilité de l'application.

Il est possible de développer des méthodes de conversion de type personnalisées. Exemple, convertir un type métier `Stagiaire` en type métier `Salarie`.

## 7. Propriétés et méthodes de classes

Certaines propriétés et méthodes peuvent s'appliquer en dehors de tout contexte d'instances, donc d'objet. Elles sont qualifiées de propriétés et méthodes statiques ou partagées.

Elles sont invoquées en faisant référence à la classe et non à une instance de celle-ci.

Nous pouvons en voir une première illustration avec la classe `String` :

Invocation de la méthode statique **Compare**

```
String.Compare(
  ▲ 1 sur 10 ▼ int string.Compare(string strA, string strB)
  Compare deux objets System.String spécifiés et retourne un entier qui indique
  strA: Premier System.String.
```

Invocation de la méthode d'instance **CompareTo** à partir d'une instance de chaîne.

```
string s1 = string.Empty;  
s1.CompareTo(|
```

▲ 1 sur 2 ▼ int string.CompareTo(object value)

Compare cette instance avec un System.Object spécifié et indique si cette instance précède,  
*value: System.Object dont la valeur est un String.*

## 7.1. Propriétés de classe

Jusqu'à présent, les propriétés déclarées étaient des **propriétés d'instance**. C'est à dire que les propriétés de chaque objet, instancié à partir de la même classe, sont porteuses de valeurs spécifiques à l'objet considéré (Nom, DateNaissance, ).

Supposons que nous souhaitions connaître le nombre d'instances (d'objets) créés à partir de la classe Stagiaire. Nous allons implémenter une nouvelle propriété qui représentera le nombre d'instances de la classe stagiaire. Cette propriété aura pour la classe une seule valeur partagée et non une valeur par instance.

En C# il est possible de créer des propriétés de classe. Leur valeur est partagée par toutes les instances d'une même classe.

Comme pour les autres propriétés, il est nécessaire de créer les méthodes d'accès associées. Pour ce compteur, seule la méthode **Get** est nécessaire. L'écriture, une incrémentation de 1, se fera dans les méthodes des constructeurs.

Pour déclarer une telle propriété, on utilise le mot-clé **static**. Dans d'autres langages comme VB, ces propriétés seront déclarées **shared**

Par exemple, dans la classe Stagiaire, le compteur d'instance pourrait être déclaré comme suit :

```
static int _compteurInstances = 0;  
  
public static int CompteurInstances  
{  
    get { return Stagiaire._compteurInstances; }  
}
```

La **propriété de classe** Compteur, initialisée à 0 lors de sa déclaration, sera incrémentée de 1 dans tous les constructeurs développés pour la classe Stagiaire.

Sa valeur sera le nombre d'instances valides à un instant donné.

**Il s'agit nécessairement d'une propriété de classe et non d'instance.**

Le **champ** et la **propriété** sont **statiques**.

Il n'est pas possible de manipuler un champ d'instance dans une propriété de classe.

## 7.2. Méthodes de classe

De même que les propriétés, certaines méthodes peuvent être définies comme statiques. Nous aurions pu implémenter une méthode de classe pour renvoyer le nombre d'instances en cours plutôt que d'implémenter cette fonctionnalité sous forme d'un accesseur Get de propriété.

Une méthode de classe est déclarée avec le mot-clé **static**. Pour avoir accès à la valeur du compteur d'instances nous pourrions déclarer une méthode GetNombresInstances() ainsi :

```
/// <summary>
/// Affichage du nombre d'instances
/// </summary>
/// <returns>Nombre instances en cours</returns>

public static int GetNombreInstances()
{
    return _compteurInstances;
}
```

Exemple courant de méthode de classe implémentée, la méthode permettant la comparaison de deux instances d'objets de cette classe.

```
public static int Compare(Stagiaire stagiaire1, Stagiaire stagiaire2)
{
    if (stagiaire1 == stagiaire2) return 0; else return -1;
}
```

Une méthode de classe, **static**, ne peut manipuler que des **champs** définis comme **static**.

## 7.1. Classes statiques

Certaines classes peuvent être définies comme statiques. Il s'agit le plus souvent de classes « boîte à outils » ou de mécanismes disponibles sous forme d'une entité unique comme la console système de Windows.

*Si une **classe** est définie comme **static**, tous ses membres doivent être **statiques**.*

```
static class ClasseStatique
```