Cégep de Saint-Hyacinthe
Département d'informatique

Programmation orientée objet
420-2DP-HY
(3-3-3)

Notions de base sur les objets - (classe, champs, méthodes, ...)

(Version 1.3)

3 à 6 heures

Préparé par
Martin Lalancette

Comprendre les éléments suivants :

- Un objet
- Programmation orientée-objet
- Les variables de type référence
- Instanciation d'un objet
- Accès, champs, méthodes, propriétés

Table des matières

Introduction

Cette séquence a pour but de vous initier aux notions de base nécessaires à l'introduction de la programmation. Nous commencerons par énoncer les éléments théoriques appuyés d'exemples simples et faciles à reproduire. Afin d'axer l'attention sur la compréhension de ces notions, il y aura des exercices à faire tout au long de cette séquence. Celle-ci portera principalement sur l'**initiation** aux **notions d'objets**. Pour bien suivre les instructions qui vont être mentionnées tout au long des séquences d'apprentissage, une préparation de base s'impose. Il est important de créer un répertoire de travail (sur votre C : ou clé USB). Voici une suggestion d'arborescence :

Préparation: S'assurer d'avoir créé l'arborescence ici haut mentionnée sur votre C ou votre clé USB. Copier ce fichier dans le répertoire rouge ici haut mentionné.

Qu'est-ce qu'un objet?

Dans la philosophie de la programmation orientée-objet, un objet (logiciel) est conceptuellement similaire à un objet présent dans le monde réel.

Voici comment nous pouvons décrire un objet :

- Il **effectue des tâches** (dans la plupart des cas)
- Il **remplit un rôle** (comble un besoin). Il a un but précis.
- Il **possède des propriétés** et des **états** qui lui sont propres (par rapport à d'autres exemplaires du même type)
- Il possède des fonctionnalités.
- Il **forme un tout** (possède un début et une fin).
- Le **type d'un objet** permet sa **classification** pour le retrouver parmi une multitude de multitudes d'autres types d'objets existants!
- Il possède une enveloppe. Il **dévoile ce qui est accessible de l'extérieur**. Tout le nécessaire… Mais, rien que le nécessaire…
- Il peut être **un contenant** pour d'autres objets.
- Il peut être **un assemblage** fait d'autres objets.
- Il peut **provenir (dériver) d'une famille** d'objets, ayant des points communs.

Un objet est connu sous le nom d'une **instance d'une classe**.

(diagrams.net/draw.io, 2022)

Qu'est-ce qu'une classe?

Une classe, c'est quoi?

- Une classe **est le plan, le modèle ou la définition** à partir desquels chaque objet individuel est créé.
- Chaque classe est une définition d'un nouveau type de données.

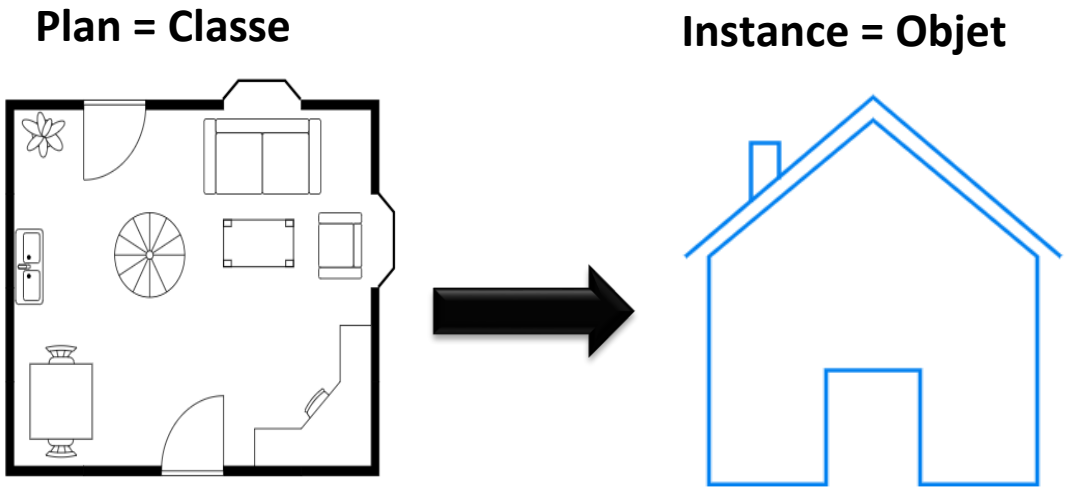


Figure 1 - (diagrams.net/draw.io, 2022)

Exemples en C#:

```
class Voiture
{
      //....
}
```

```
class Telephone
{
      //....
}
```

```
class Ordinateur
{
      //....
}
```

```
class Maison
{
      //....
}
```

La programmation orientée-objet

Les langages de haut niveau (Java, C#, VB.Net) proposent une programmation dite **orientée-objet**, c.-à-d. une programmation basée sur la façon de concevoir des objets physiques (conceptuellement) de la vraie vie. Exemples : automobile, avion, radio, etc. Chacun de ses exemples possède des **états** et des **comportements**. Prenons le cas de l'automobile. Elle possède plusieurs états : son modèle, sa couleur, sa vitesse courante, son niveau d'essence et j'en passe. Elle possède également plusieurs comportements : l'accélération, le freinage, le changement de vitesse, etc. Une automobile est un objet :

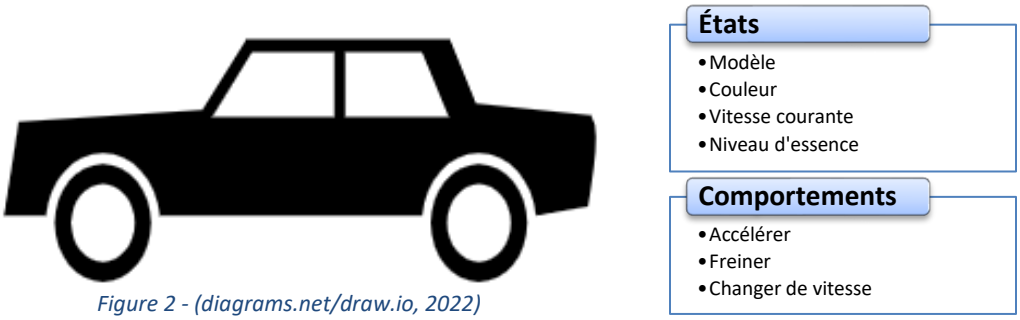
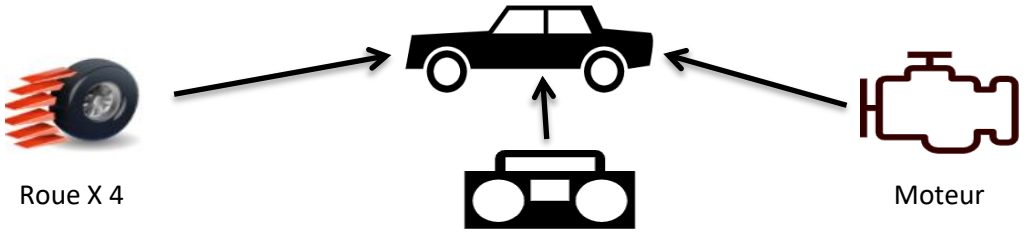


Figure 2 - (diagrams.net/draw.io, 2022)

Certains objets sont simples, d'autres sont plus complexes. Il est évident qu'une automobile est un objet complexe. **Dans le cas des objets complexes, ils doivent être décomposés en plus petits objets ayant leurs propres états et comportements**. Voici un exemple avec l'automobile :



(diagrams.net/draw.io, 2022)

Dans la conception d'un programme orienté-objet, l'objet emmagasine ses états dans des **champs** (variables globales) ou **propriétés** et expose ses comportements via des **méthodes**. Une classe peut posséder de plusieurs sortes de membres :

- les champs (variables)
- les méthodes (tâches)
- les constantes
- les propriétés (autogénérées)
- les événements
- les indexeurs
- les opérateurs
- les constructeurs et le destructeur
- les types internes

Nous aborderons chaque membre en temps et lieu.

Encapsulation

Lorsque nous faisons appelle à un membre (méthode, propriété, …) d'un objet et que nous obtenions un résultat sans connaître la complexité de ce membre, ce concept qui cache cette complexité s'appelle l'**encapsulation**. En d'autres termes, de l'extérieur, les objets rendent visible ce qui est utile, et dissimulent ce qui ne l'est pas. Ceci est pour des raisons pratiques, sécuritaires, et aussi confidentielles.

Événement

Un événement est une surveillance de la réalisation d'un fait (d'une condition) à communiquer à ceux qui veulent y donner suite. Que fait un objet avec un événement :

- Certains objets **déclenchent** des événements. *Ex. : un réveil matin*
- Certains objets **réagissent** à des événements. *Ex. : une boîte vocale*
- Certains objets ont une **combinaison des deux**. *Ex. : un système anti-vol (appel à la centrale*, activer les sirènes*, démarrer la caméra vidéo)*.

Allocation mémoire « managed »

La mention « managed » signifie que la mémoire est gérée par une entité dont le rôle est de s'assurer de la bonne gestion de la mémoire, communément appelée « Garbage collector ». Mais avant, voici la description de deux types d'allocations mémoire.

Allocation sur la pile (stack)

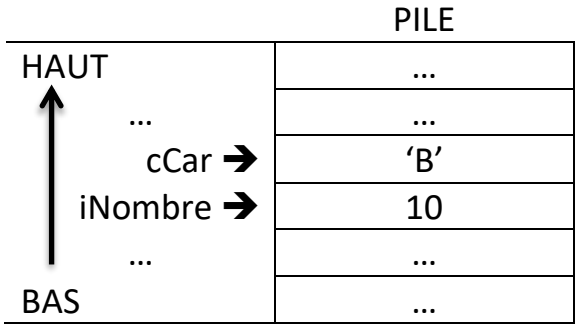
L'exécution d'un programme tourne autour d'une pile (zone mémoire souvent appelée *Callstack*) qui définit les règles d'appel pour les éléments suivants :

- Les variables locales
- Les paramètres
- Adresse et valeur de retour d'une fonction/procédure/méthode

La pile croît à chaque appel de méthodes et décroît à chaque retour. Le tout est géré automatiquement par l'environnement d'exécution (runtime).

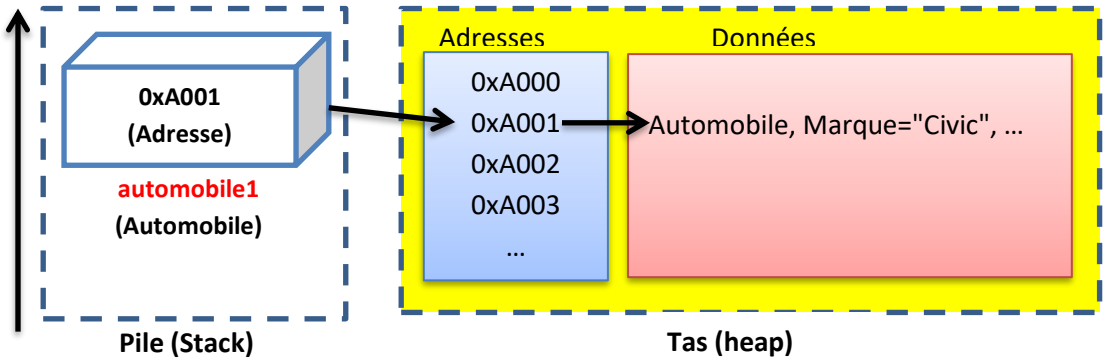
Exemple : Exemple avec des variables locales.

```
int iNombre = 10;
char cCar ='B';
```



Allocation sur le tas (heap)

Le tas est la mémoire disponible pour un programme, au moment de son exécution, afin d'allouer ou désallouer dynamiquement cette mémoire sur demande. Il peut y avoir des fuites de mémoire (memory leak), le programmeur doit être prudent. Cependant, en C# grâce au Framework, il y a une entité qui facilite la tâche du programmeur qui est le GarbageCollector. Il s'occupe de désallouer le mémoire au moment opportun, surtout lorsque les objets ne sont plus utilisables. **Les objets (créés à partir de classes) se trouvent automatiquement créés sur le tas**. Exemple :

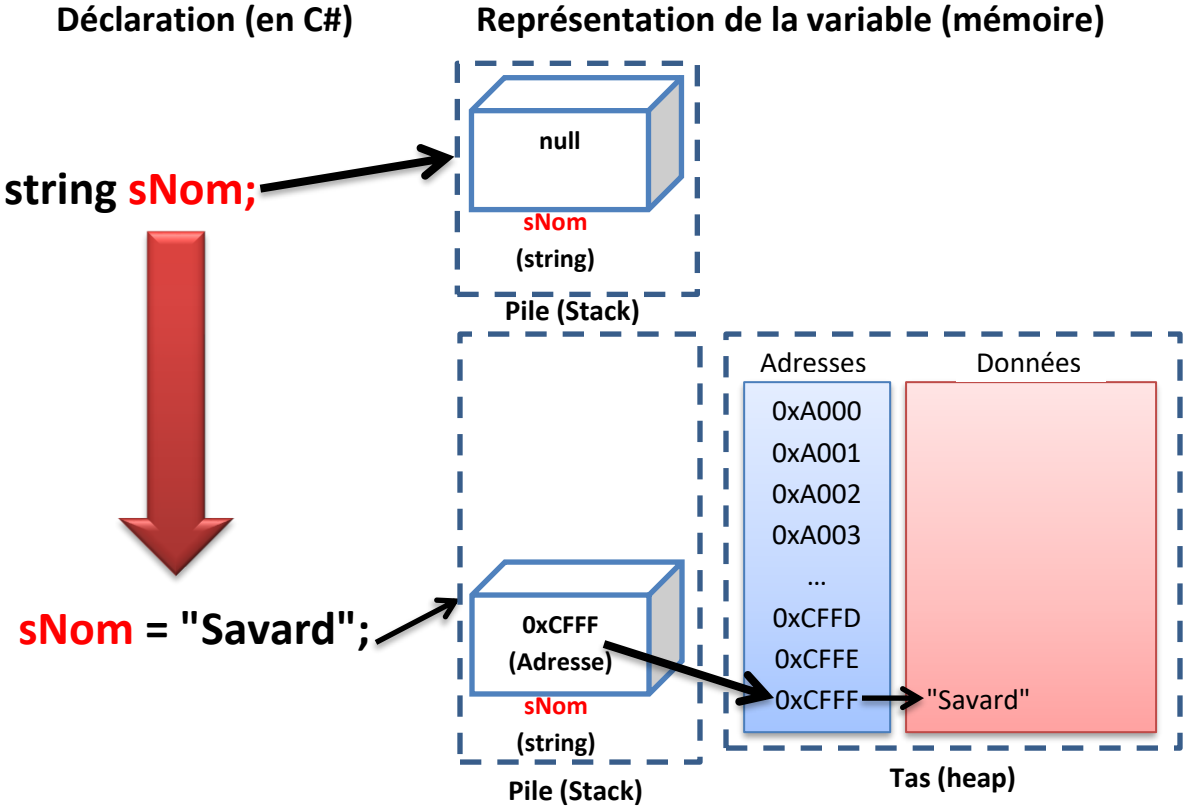


Garbage Collector (GC)

Le GC a comme rôle de gérer l'allocation et la libération de la mémoire utilisée par une application. Tant qu'il y a de la place sur le tas, le GC alloue la mémoire. Cependant la mémoire n'est pas infinie. Le GC se doit de s'exécuter afin de libérer de la mémoire. C'est le moteur du GC qui détermine le meilleur moment pour le faire et en fonction du nombre d'allocations en cours. Le principe est de vérifier s'il existe des objets dans le tas qui ne sont pas utilisés (ex. : dont aucune variable ne pointe dessus). Donc l'action de mettre à **null** une variable de type référence permet d'indiquer au GC de libérer la mémoire de cet objet.

Les variables de type référence

Maintenant, nous allons voir qu'une variable de type référence ne va pas emmagasiner la valeur, mais plutôt l'**adresse** qui indique où la valeur sera localisée en mémoire dite tas. Les types références incluent les types créés par les mots clés : *object*, *string*, *interface*, *delegate*, et *class*. Comment ça fonctionne en mémoire :



À savoir :

- Comme toute variable, son type fait en sorte qu'elle accepte que les valeurs qui seront du même type ou d'un type dérivé (le type dérivé sera expliqué dans le cours sur l'héritage)
- C# doit connaître nos besoins en variables à l'avance. La création d'une variable permet de réserver et de préparer la mémoire en conséquence.

Après avoir traité la déclaration, C# a effectué les étapes suivantes pour préparer la variable :

1. il lui a trouvé un endroit en mémoire (à une adresse)
2. il a réservé l'espace dont elle a besoin (selon le type),
3. et il y a affecté la valeur null par défaut (tous les types références).

À savoir :

- De l'extérieur, une variable de type référence (ou simplement **variable référence**) est très similaire à une variable de type valeur.
- Au lieu de contenir une valeur ordinaire, une variable référence contient l'emplacement (l'adresse) d'un objet qui est en mémoire, quelque part. Autrement dit, elle est la référence d'un objet.
- Si une variable référence ne réfère à aucun objet, elle contient la valeur null (comme rien ou nulle part).

Instanciation d'un objet

Petit rappel → une classe est :

- Unique. **C'est pourquoi on dit qu'une classe est un type**.
- Une classe est un plan servant à construire des objets.
- Chaque objet est du type de la classe qui a servi à le créer.

L'opération qui permet de créer un objet en mémoire à partir d'une classe s'appelle : **l'instanciation**. Les instanciations utilisent l'opérateur **new**.

Le C# permet d'utiliser une variable référence selon 3 modes :

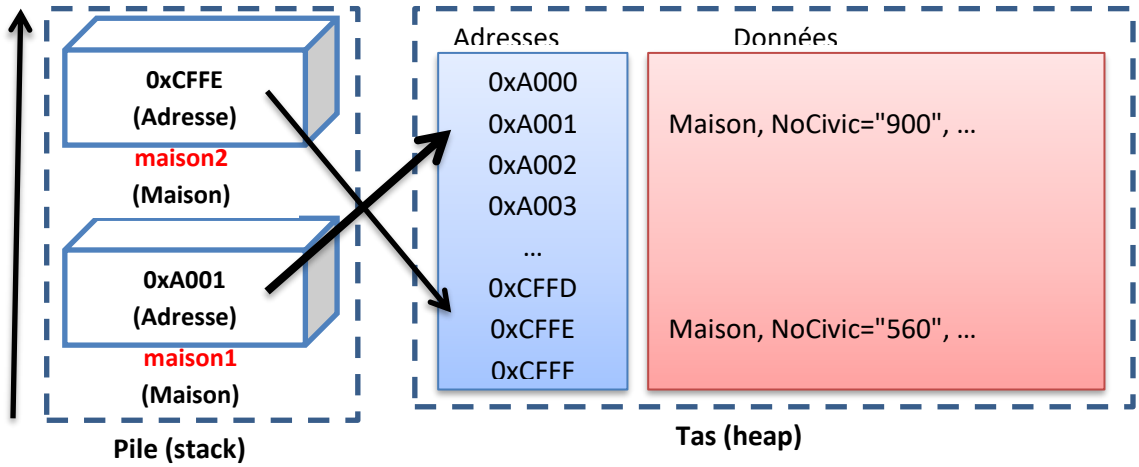
- L'**affectation** : consiste à écrire l'adresse d'un objet, dans la mémoire de la variable référence.
- La **lecture** : consiste à lire l'adresse d'un objet, mémorisée par la variable référence.
- L'**accès** (variable référence suivit du « . ») : consiste à utiliser l'objet
- situé au bout de l'adresse, mémorisée par la variable référence.

Exemple #1: Créer deux objets de type Maison.

```
Maison maison1 = new Maison();
maison1.NoCivic = "900"; // Définition d'une propriété que nous verrons plus loin.

Maison maison2 = new Maison();
Maison2.NoCivic = "560";

Console.WriteLine(maison1.NoCivic);
Console.WriteLine(maison2.NoCivic);
```



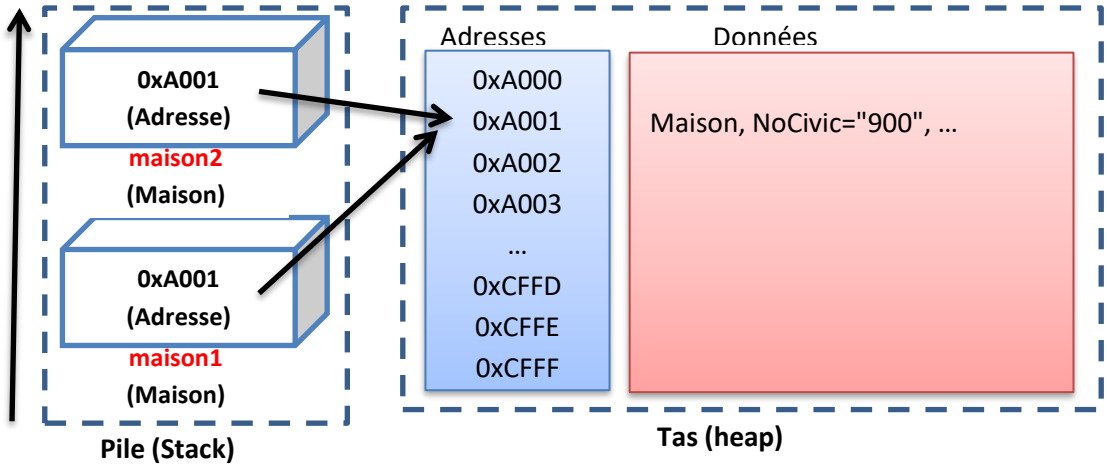
Ici, nous avons donc **deux variables de type référence dans la pile** qui pointent sur **deux objets distincts dans le tas**.

Exemple #2: Créer un objet de type Maison avec deux variables qui pointent sur le même objet.

```
Maison maison1 = new Maison();
maison1.NoCivic = "900";

Maison maison2 = maison1;
Console.WriteLine(maison1.NoCivic);
Console.WriteLine(maison2.NoCivic);
```

ATTENTION! Même référence donc même objet!!



Survol d'une définition d'une classe toute simple

Une classe ***est le modèle (ou plan) à partir duquel des objets individuels sont créés***. Dans la vie de tous les jours, les objets ont besoins de modèle ou plan pour définir comment ils doivent être construits. La classe le permet du côté informatique. Pour illustrer simplement la conception d'une classe, j'utiliserai le rectangle comme objet, sa hauteur et sa largeur comme des états et l'obtention de la surface comme comportement. Voici le résultat en code C# :

```
class Rectangle
{
    int _iHauteur;
    int _iLargeur;

    int CalculerSurface()
    {
        return _iHauteur * _iLargeur;
    }
}
```

Définition de l'objet (c.-à-d. une **classe**)

Définition des états (c.-à-d. les **variables** ou **champs**)

Définition d'un comportement (c.-à-d. une **méthode**)

Définition des actions (c.-à-d. l'**algorithme**)

Définir les accès aux membres grâce aux modificateurs

Lors de la définition d'une classe, il est important d'y indiquer, pour chaque membre de cette classe (ex. : propriétés, méthodes, constructeurs, constantes, etc.) leurs niveaux d'accès. Par exemple, si l'on veut que les variables globales ne soient pas accessibles de l'extérieur de l'objet, il faut les déclarer avec le modificateur *private*. ***En passant lorsque l'on omet de dé-***

finir l'accès pour un membre, c'est le modificateur <u>private</u> qui est appliqué par défaut. Voici un tableau qui mentionne les types d'accès possibles :

Modificateur	Signification
private	Seul le code de la classe elle-même peut avoir accès aux membres. Invisible de l'extérieur.
public	Tout autre code du même assembly (c.-à-d. : programme ou une librairie) ou non, qui fait référence à l'objet peut accéder au membre.
~~**protected**~~	Seul le code de la classe elle-même ou **<u>d'une classe dérivée de celle-ci</u>** peut avoir accès aux membres.
~~**internal**~~	Tout code du même assembly, mais pas d'un autre assembly, peut accéder au membre.
~~**internal protected**~~	Le type ou le membre est accessible par tout code de l'assembly dans lequel il est déclaré, ou à partir d'une classe dérivée dans un autre assembly

Dans cette séquence, nous allons nous attarder principalement aux deux premiers soit ***private*** et ***public***. Les autres nécessitent des notions plus avancées qui seront vues en deuxième session.

Revenons avec l'exemple de Rectangle, les deux déclarations suivantes sont équivalentes :

```
class Rectangle
{
    int _iHauteur;
    int _iLargeur;

    int CalculerSurface()
    {
        return _iHauteur * _iLargeur;
    }
}
```

Équivalent

```
public class Rectangle
{
    private int _iHauteur;
    private int _iLargeur;

    private int CalculerSurface()
    {
        return _iHauteur * _iLargeur;
    }
}
```

```
Rectangle r = new Rectangle();
r.
```

- Equals
- GetHashCode
- GetType
- ToString

Étant donné que tous les membres de la classe sont *private*, ils ne sont pas disponibles. Pas très pratique dans ce cas!!!

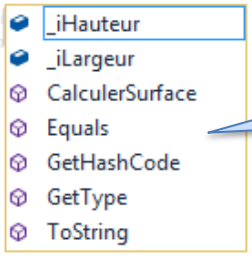
Donc lors de la définition il faut prévoir quels seront les membres qui seront accessibles. Dans l'exemple précédent je peux les rendre tous accessibles grâce à ***public*** :

```
public class Rectangle
{
    public int _iHauteur;
    public int _iLargeur;

    public int CalculerSurface()
    {
        return _iHauteur * _iLargeur;
    }
}
```

Cependant, il n'est pas souhaitable de rendre des variables globales publiques, car elles sont destinées au fonctionnement interne de cette classe.

```
Rectangle r = new Rectangle();
```

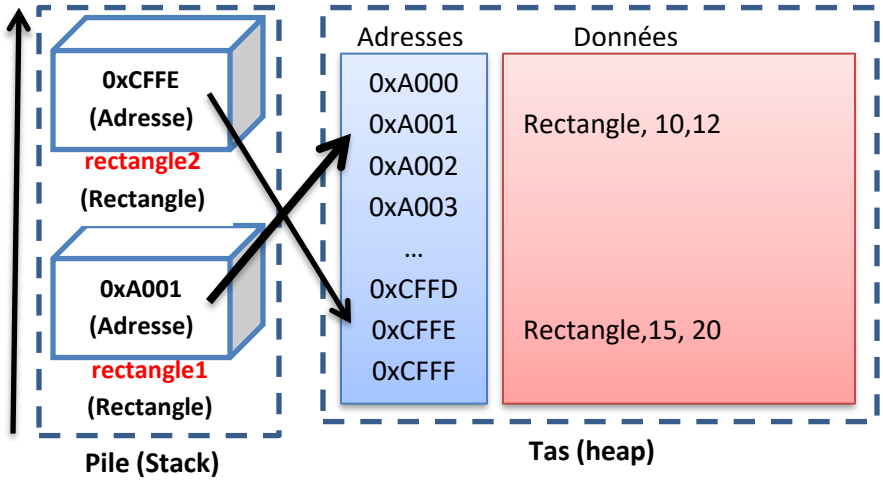


Nous constatons maintenant que les trois membres sont visibles et utilisables.

Exemple #1: Créer deux objets de type Rectangle.

```
Rectangle rectangle1 = new Rectangle(); // Création du premier objet.
rectangle1._iHauteur = 10;
rectangle1._iLargeur = 12;

Rectangle rectangle2 = new Rectangle(); // Création du deuxième objet.
rectangle2._iHauteur = 15;
rectangle2._iLargeur = 20;
```



Ici, nous avons donc **deux variables de type référence dans la pile** qui pointent sur **deux objets distincts dans le tas**.

Exercice 1. :	Créer la solution et un projet de type console nommé **InitiationClasses**. Dans ce projet, une classe nommée **Rectangle** dans un dossier **Models** (MVVM). Y définir les champs **Hauteur** et **Largeur** ainsi que la méthode **CalculerSurface**. Dans le programme principal, des questions doivent être posées pour remplir deux objets Rectangle et afficher la surface des deux rectangles.

Exercice 2. :	Ajouter au projet, une classe nommée **Étudiant**. Créer les champs **publics** suivants et déterminer leurs types : Nom, Prénom, Numéro de DA et la date de naissance. Instancier deux objets étudiants avec les données suivantes : - Tremblay, Carl, 0234789, 1996-06-09 - Roy, Line, 1212908, 1997-05-23

Exercice 3. :	À la classe **Etudiant**, ajouter un champ public qui contiendra un tableau de 5 notes associées à l'étudiant. Ajouter une méthode nommée « **CalculerMoyenne** » qui effectue la moyenne de l'ensemble des notes.

Chose importante à savoir concernant une bonne pratique de programmation : **l'utilisation de variables globales (champs) est fortement suggérée pour la gestion interne de la classe. Il faut éviter de pouvoir modifier sa valeur en plein traitement de l'extérieur de la classe**. Donc, **il faut toujours déclarer les champs avec le modificateur** ***private*** pour éviter leurs accès de l'extérieur. Il est conseillé d'utiliser **des méthodes** ou bien de définir **des propriétés** pour y déposer ou changer leurs valeurs.

Définir une méthode

Les notions associées à la déclaration d'une méthode ont été vues dans la séquence intitulée « Notions de base sur les objets – Les méthodes ».

Exercice 4. : Vu qu'il n'est pas conseillé de donner accès aux champs directement d'une classe, changer les modificateurs pour **privé**. Concevoir les méthodes suivantes concernant la classe **Étudiant**:

DefinirInfos : ayant comme paramètres : nom, prénom, numéro DA et année de naissance. Remplir les champs.

AttribuerNote : ayant comme paramètre : la note et la position (entre 1 et 5). Affecter la note dans le tableau selon la position (ajuster en fonction de l'index). Valider les limites.

ObtenirSommaire : sans paramètre, elle retourne une chaine de caractères contenant les informations de l'étudiant. Exemple :

```
Line Roy (1212908), date de naissance : 1997-05-23, Moyenne : 67.6
```

Exercice 5. Ajouter une classe nommée **Exercices**. Écrire la méthode « **ObtenirMois** » qui, à partir d'un nombre compris entre 1 et 12, retourne le mois correspondant (1=Janvier, 2=Février, etc.). Prévoir l'entrée d'un nombre invalide. Afficher le mois retourné.

Exercice 6. Dans la classe **Exercices,** écrire la méthode « **ObtenirMention** » qui à partir d'une lettre (majuscule ou minuscule) retourne la mention « BAS » si l'utilisateur entre A, B, C ou D et la mention « HAUT » si l'utilisateur entre E, F, G ou H. Dans les autres cas, afficher « ERREUR » comme mention sinon la mention retournée.

Exercice 7. : Dans la classe **Exercices,** écrire la méthode « **InverserLettresPhrase** ». Elle doit prendre en paramètre une chaine de caractères (ex. : « Vive la programmation! », inverser la phrase et retourner le tout dans **un tableau 1D de type *char***.

Définir des propriétés à une classe

Les propriétés vous permettent d'accéder aux données de la classe de façon flexible et sécuritaire. Elles sont des membres de la classe qui peuvent être accessibles comme un champ de données, mais peuvent contenir du code comme une méthode pour mieux exercer un contrôle. Les propriétés ont de multiples utilisations :

- Elles peuvent valider des données avant d'autoriser une modification
- Elles peuvent exposer de façon transparente sur une classe des données récupérées d'une source quelconque, telle qu'une base de données
- Elles peuvent effectuer une action à la suite d'une modification de données, par exemple déclencher un événement ou modifier les valeurs d'autres champs.

Une propriété a deux accesseurs (accessors en anglais):

Accesseur	Description
get	L'accesseur *get* est utilisé pour **retourner la valeur de la propriété**.
set	L'accesseur *set* est utilisé pour **affecter une nouvelle valeur à la propriété**.

Lorsque vous définissez une propriété de cette manière, vous n'êtes pas obligé d'ajouter le type dans le nom. Voici comment déclarer la propriété Hauteur dans la classe Rectangle basée sur la variable _iHauteur:

```
public int Hauteur
{
    get
    {
        return _iHauteur;
    }
    set
    {
        _iHauteur = value;
    }
}
```

Equivalent

```
public int Hauteur
{
    get => _iHauteur;
    set => _iHauteur = value;
}
```

Utilisation de lambda!!!

value est un mot réservé qui représente une valeur qui a été affectée à la

propriété. Nous avons la possibilité de valider le contenu avant de l'affecter à la variable globale. Exemple :

```
public int Hauteur
{
    get
    {
        return _iHauteur;
    }
    set
    {
        if (value > 0) // On accepte seulement les valeurs positives.
            _iHauteur = value;
    }
}
```

Cette technique offre aussi le choix de permettre ou non l'accès indirect au champ privé, mais aussi d'effectuer des validations ou des traitements avant de procéder.

Une propriété peut posséder 1 des 2 accesseurs : **get** ou **set**. Dans la très grande majorité des cas, c'est l'accesseur **get** que l'on retrouve seul. À savoir :

- Lorsqu'un **type valeur** est retourné, c'est une copie de la valeur qui est retournée et non la valeur d'originale.
- Lorsqu'un **type référence** est retourné, c'est une copie de la référence qui est retournée. On a donc accès au même objet référé.
- Les accesseurs **get** et **set** possèdent le même niveau d'accès que celui défini au niveau de la déclaration de la propriété.
- Un modificateur d'accès différent de celui de la propriété peut être explicitement déclaré pour 1 des 2 accesseurs. Cependant, le modificateur d'accès devra être plus restrictif que celui déclaré au niveau de la propriété.

  Exemple :

Permis :

```
private byte _byAge;    // Champ
public byte Age         // propriété
{
    get { return _byAge; }
    private set { if (value >= 0)
                    _byAge = value; }
}
```

Non permis :

```
private byte _byAge;    // Champ
private byte Age        // propriété
{
    get { return _byAge; }
    public set { if (value >= 0)
                    _byAge = value; }
}
```

- Dans la très grande majorité des cas, c'est l'accesseur **set** qui est doté du modificateur d'accès explicite **private**.
- **Remarque importante** : les accesseurs ne sont pas utilisés exclusivement pour l'encapsulation des champs. Très souvent, les propriétés sont aussi utilisées par les autres membres de la classe afin d'assurer un contrôle adéquat sur un champ.
  Exemple : le champ _byAge ne devrait jamais être négatif. Mais quelque part dans le code, on pourrait **accidentellement** contourner cette règle en accédant directement au champ!
- Les propriétés, comme les méthodes, facilitent l'évolution des classes, car les clients programmeurs n'ont qu'à se soucier des appels et du type de retour, et non de la logique utilisée par les propriétés et les méthodes.

Une façon simple d'englober une variable globale pour en faire une propriété est de faire le code snippet suivant:

Snippet	Description	Exemple

propfull	déclaration d'une propriété traditionnelle	`private int myVar;`
		`public int MyProperty`
		`{`
		`  get { return myVar; }`
		`  set { myVar = value; }`
		`}`

Exercice 8. : Transformer la variable **_iLargeur** en propriété nommée **Largeur** et la variable **_iHauteur** en **Hauteur** de la classe **Rectangle**.

Exercice 9. : Coder la classe suivante :

- Définir la classe qui va représenter une **Personne** ayant comme propriétés son nom (type string), son prénom (type string), et son âge (type int). Créer une méthode (nommée CalculerDuréeVieEnHeures) qui permet de retourner le nombre d'heures de vie de cette personne.

- Créer les trois personnes (objets) suivantes :

1. Marc Tremblay 34 ans
2. Ginette Roy 26 ans
3. Luc Savard 43 ans

- Appeler la méthode CalculerDuréeVieEnHeures pour chaque personne et afficher le résultat.

- Afficher le nombre d'heures (Durée de vie) total.

Exercice 10. : Ajouter à la classe **Personne** une propriété **Téléphone** qui acceptera les formats suivants : (450) 123-1234, 450-123-1234, mais qui devra retirer les caractères de trop pour conserver l'essentiel. Exemple : 4501231234.

Exercice 11. : Coder la classe suivante :

- De définir la classe qui va représenter une **Automobile** ayant comme propriétés son modèle (type string), son fabricant (type string), son année (type int) et son kilométrage actuel (type int). Créer une méthode (nommée MoyenneKmParAnnée) qui permet de retourner le nombre de kilomètres par année pour cette automobile.

- Créer les deux automobiles (objets) suivantes :

1. Sonata de Hyundai 2010 avec 30000 km (a1)
2. CRV de Honda 2008 avec 64500 km (a2)

- de créer deux variables (a3 et a4) de type référence supplémentaire qui pointent sur chacune des voitures.

- Afficher la moyenne totale de km des deux voitures.

Exercice 12. : Ajouter une propriété traditionnelle nommée **Vitesse**. Valider que la vitesse ne soit pas négative et qu'elle ne dépasse pas 120 km/h.

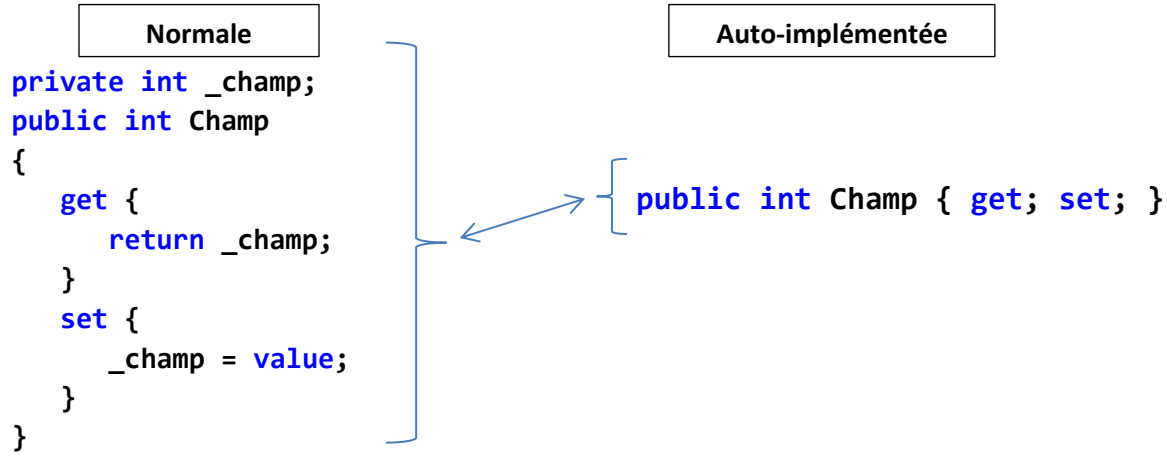
Pourquoi ne pas utiliser un champ public au lieu d'une propriété?

Cette question vient souvent à l'esprit. Voici quelques réponses :

1. Rendre un champ public permet de le rendre disponible à l'extérieur de la classe. C'est vrai, mais il n'y a pas de contrôle sur ce qui sera déposé dans ce champ. Une mauvaise valeur pourrait faire mal fonctionner votre logique et même faire planter votre programme (exemple : *null*).
2. Souvent, nous avons besoin de rendre l'information disponible à l'extérieur sans toutefois permettre de la modifier. Rendre un champ public, c'est rendre accessible l'information contenue dans ce champ, mais également rendre accessible sa modification. Il est préférable alors d'utiliser une propriété en rendu publique l'accesseur *get* et de priver le *set*.

Définir des propriétés auto-implémentées

C# a introduit dès la version 3 du *framework* une façon de simplifier la déclaration d'une propriété lorsque celle-ci ne nécessite pas d'algorithmes additionnels (par exemple sans avoir de validation) dans les accesseurs **get** et **set**. En voici la forme :



Où est le champ? Le champ de ce type de propriété est défini implicitement, c.-à-d. que le programme va créer un champ en mémoire non visible et accessible par le code pour y déposer la valeur. Son contenu sera donc accessible seulement par la propriété.

Snippet	Description	Exemple
prop	déclaration d'une propriété auto-implémentée.	`public int MyProperty { get; set; }`
propg	prop avec le modificateur d'accès private à l'accesseur set.	`public int MyProperty { get; private set; }`

Il est facile de « bouder » les propriétés à cause de la sensation d'encombrement qu'elles procurent, surtout pour une nouvelle classe. Cependant, l'utilisation et l'importance des propriétés sont réelles et omniprésentes dans C#. Exemple avec **Rectangle** sans validation :

```
class Rectangle
{
    // Priopriétés auto-implémentées
    public int Hauteur { get; set; }
    public int Largeur { get; set; }
}
```

Exercice 13. : Ajouter les membres suivants à la classe Automobile :

1) **Propriétés auto-implémentées**: Nom, Couleur, ToitOuvrant.

2) Créer trois instances.

Exercice 14. : Dans le projet de l'exercice précédent, ajouter une classe appelée **Roue** qui va s'occuper de gérer les informations associées à la jante et au pneu d'un véhicule. Voici un exemple d'information : **195/65 R 15 91 H M+S**. Pour chacune des informations, déclarer une propriété ou une propriété auto-implémentée :

- Largeur du pneu gonflé en millimètre. Ex. : 195
- Série (hauteur du flanc en %), Ex : 65
- Type (R=Radial, B=Carcasse BIAS, D=Diagonale). Ex. : R
- Diamètre de la jante en pouces (>0). Ex. : 15
- Indice de capacité de charge (entre 50 et 120). Ex. : 91
- Code de vitesse en km/h (Q=160, R=170, S=180, T=190, U=200, H=210, VR>210, V=240, ZR>240, W=270, Y=300. Ex. : R
- Signe spécial : M+S (Mud+Snow = boue et neige = Hiver)
- Marque du pneu.

Ajouter une propriété **Etiquette** qui génère une chaine de caractères sous forme « **195/65 R 15 91 H M+S** » après des propriétés précédentes.

Bibliographie

diagrams.net/draw.io. (2022, 01 05). *Usage terms for diagrams created in diagrams.net*. Récupéré sur diagrams.net: https://www.diagrams.net/doc/faq/usage-terms