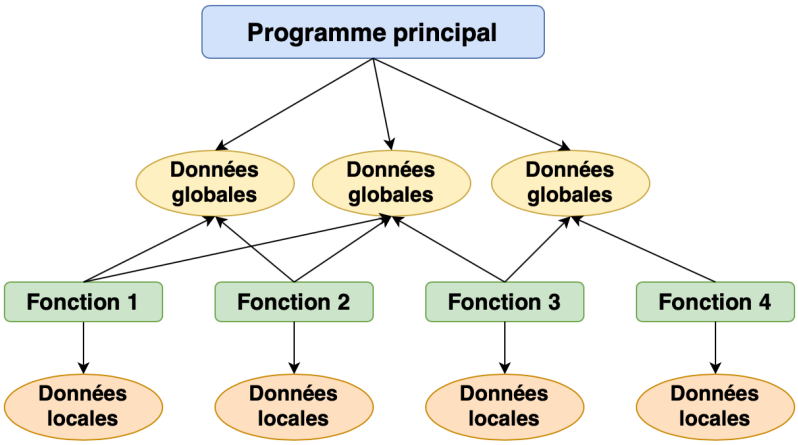
Programmation Orientée Objet	
Classe :	Terminale NSI, classe entière
Capacités attendues :	-Écrire la définition d'une classe -Accéder aux attributs et aux méthodes d'une classe

I - Une nouvelle façon de programmer

1 - De la programmation procédurale à la programmation orientée objet

Jusqu'ici, tous nos programmes ont été réalisés en **programmation procédurale**. C'est un paradigme de programmation basé sur le concept d'appel procédural. Une procédure, aussi appelée *routine*, *sous-routine*, ou *fonction* contient simplement une série d'étapes à réaliser. N'importe quelle procédure peut être appelée à n'importe quelle étape de l'exécution du programme, incluant d'autres procédures voire la procédure elle-même (récursivité).



Nous remarquons que les fonctions, procédures et autres suites d'instructions opèrent sur leurs données locales ainsi que sur des données globales partagées entre elles.

```
def afficher_liste(liste_course):
    '''Affiche le contenu de la liste des courses'''
    if (liste_course == []):
        print("-----La liste de courses est vide")
        return
    print("Le contenu de votre liste de courses: ")
    for indice, element in enumerate(liste_course):
        print(indice+1,"-",element)

def ajouter_element(liste_course, nouveau_element):
    '''Ajouter le nouveau_element à la fin de la liste des courses'''
    liste_course.append(nouveau_element)
    print("-----L'élément ", nouveau_element, "a bien été ajouté "+
          "à la liste de courses")

#Programme principal
if __name__ == "__main__":
    liste = []
    ajouter_element(liste,"Pomme")
    afficher_liste(liste)
```

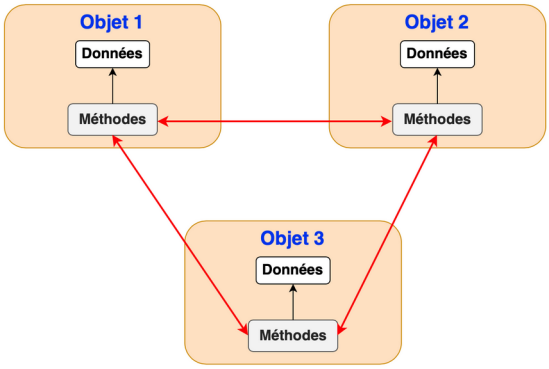
C'est une programmation efficace pour les débutants et tant qu'une seule personne travaille sur un programme qui n'est pas gros. Cependant, nous remarquons qu'il y a donc une **dissociation** entre

les données et les fonctions ce qui pose des difficultés lorsque l'on veuille représenter des objets de monde réel. De plus, les procédures s'appellent entre elles et peuvent donc agir sur les même données provoquant ainsi des effets de bord. Nous remarquons également que les données ne sont pas protégées des modifications (car la plupart d'entre elles sont globales). Au fur et à mesure du codage, le code devient volumineux et la maintenance devient relativement difficile (on peut avoir un code *spaghetti*).

Lorsqu'il s'agit de réaliser de gros projets logiciels où plusieurs programmeurs travaillent simultanément, il est nécessaire de programmer autrement afin d'éviter les conflits entre les fonctions construites par chaque personne. Chaque programmeur conçoit une partie du programme (un module avec des données et des fonctions documentées) qui sera assemblée avec autres parties pour réaliser une application logicielle.

On définit alors une approche de programmation basée sur des entités de base appelés **objets.** C'est ce qu'on appelle la **Programmation Orientée Objet** (POO).

Un **objet** est une **entité du monde réel** (une maison, une voiture, une personne, un compte bancaire, un animal, etc) qui possède une **identité** (adresse, référence ou nom), des **propriétés ou états** (données ou variables ou **attributs**) et des comportements (**méthodes**). Les objets communiquent entre eux à l'aide de messages (appels de méthodes) :



2 - Un peu d'histoire

La POO a fait ses début dans les années 1960, avec les réalisations dans le langage *Lisp*. Cependant, elle a été formellement définie avec les langages *Simula* (vers les années 1970) puis *SmallTalk*. Puis elle s'est développée dans les langages anciens comme le Fortran, le Cobol et devenue incontournable dans les langages les plus récents comme *Java*, *Python*.

En réalité, nous avons déjà travaillé avec des objets. Par exemple, pour le type de données *list*, une action possible est le tri d'une liste, on le fait avec **liste.sort()**. La syntaxe d'appel **nom_variable.nom_méthode()** permet d'appliquer une méthode (comportement) sur des données.

3 - Une démarche de conception orientée objet :

En 1995, ***Grady Booch*** propose 5 étapes dans l'établissement d'une conception orientée objet. Cette démarche est très utile pour un débutant. Ces étapes sont :

→ Identifier les objets du monde réel que l'on voudra réaliser et leurs attributs ;

→ Identifier les opérations ou les actions que l'objet subit de son environnement et qu'il provoque sur son environnement

→ Établir la visibilité en définissant ses relations avec les autres objets

→ Établir l'interface de l'objet avec le monde de l'extérieur (quelles fonctionnalités sont accessibles et sous quelles formes)

→ Implanter les objets en écrivant le code.

II - Vocabulaire à connaître

1 - Objet

Les objets sont omniprésents dans notre monde. Un objet peut être une entité vivante, un objet physique ou même un concept commercial tel qu'un compte bancaire.

Un objet = identité + état + comportements

L'**identité** doit permettre d'identifier sans ambiguïté l'objet (adresse/référence ou nom). L'**état** d'un objet est défini par ses propriétés (qui sont représentées par les variables ou les données) et les comportements sont implémentés à l'aide de méthodes (procédures / fonctions).

Exemples :

→ Modélisation d'un objet être humain

Son identité peut être un numéro ou son nom

Son état est représenté par ses données : sa taille, son poids, la couleur de ses yeux, etc

Ses comportements peuvent être : respirer, marcher, parler, etc

→ Modélisation d'un objet voiture

Son identité peut être le numéro de châssis, ou son propriétaire

Son état est représenté par ses données : son modèle, sa marque, sa couleur, etc

Ses comportements peuvent être : s'arrêter, rouler, démarrer, etc

2 - Classe

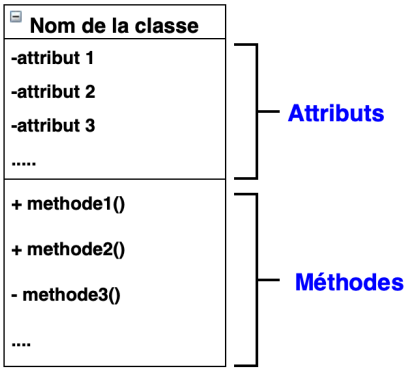
Le monde réel regroupe des objets du même type. Ainsi, il est pratique de concevoir une maquette (un moule) d'un objet et de produire les objets à partir de cette maquette. En POO, une maquette se nomme une **classe**.

Une **classe** est donc un modèle (ou un nouveau type de données abstrait) possédant des caractéristiques et des actions.

Les caractéristiques (variables ou propriétés) de la classe s'appellent **attributs**. Il peut y avoir des **attributs de la classe** et des **attributs de l'objet**. Les attributs de la classe sont définis au niveau de la classe et concernent tous les objets de la classe. Les attributs d'objets sont définis au niveau de l'objet et lui sont propres. Les actions possibles à partir des objets s'appellent **méthodes**.

En UML[1], une classe est représentée de la manière suivante :

1 **UML** pour *Unified Modeling Language* , est un langage de modélisation graphique à base de pictogrammes conçu comme une méthode dans les domaines du développement logiciel et en conception orientée objet



3 - Instance

Les objets associés à une classe se nomment des **instances**. Une instance est un objet, occurrence d'une classe, qui possède la structure définie par la classe et sur lequel les opérations définies dans la classe peuvent être appliquées.

Instancier un objet consiste dans un premier temps à lui allouer un espace mémoire, puis l'initialiser en fixant son état.

Exemples :

→ création de deux instances de la classe Point:

```
point_1 = Point()
point_2 = Point()
```

4 - Méthode constructeur

C'est un type particulier de méthode qui permet de construire l'objet désigné par la classe au moment de l'instanciation. Une méthode constructeur commence toujours par :

```
def __ init__ (self, paramètres)
```

Les paramètres permettent notamment de définir les attributs. Le mot clé **self** (un argument toujours placé en premier) désigne l'objet lui même. On pourrait l'appeler par n'importe quel nom, mais une convention très forte consiste à le nommer **self**.

Exemple :

```
 8 class Eleve:
 9     '''Un élève est caractérisé par:
10     -->ine: Identifiant national eleve
11     --> nom
12     --> et prénom'''
13
14     #Constructeur
15     def __init__(self, ine, nom, prenom):
16         self.ine = ine
17         self.nom = nom
18         self.prenom = prenom
19         print("Création réussie d'objet")
20
21 eleve = Eleve("A89786754BC", "DUPONT", "Hugo")
22 print("INE:{0}, NOM:{1}, Prénom:{2}".format(eleve.ine,eleve.nom,eleve.prenom ))
```

Analysons ce code :

→ Ligne 8 : la définition de la classe est constituée du mot-clé **class**, du nom de la classe et des deux points « : » ;

→ Lignes 9 à 12 représentent les chaînes de documentation (***docstrings***). On trouve une description del'ensemble des attributs de la classe.

→ Lignes 15 à 19 : c'est la méthode constructeur qui commence par le mot clé **def** suivi du nom__**init**__. En Python, tous les constructeurs s'appellent ainsi. Les noms de méthodes entourés de part et d'autre de deux signes soulignés **(__nom_methode__)** sont des méthodes spéciales. Notez que, dans notre définition de méthode, nous passons un premier argument nommé **self**.

Dans le constructeur, on crée des variables **self.ine**, **self.nom** et **self.prenom** que l'on initialise avec les paramètres passés au constructeur lors de l'instanciation.

→ Ligne 21 : Quand on veut créer un objet de type **Eleve**, on appelle le constructeur de la classe **Eleve**. Celui-ci prend en paramètre la variable self qui représente l'objet en train de se créer.

Conventions de nommage :
→ Une **classe** s'écrit toujours avec une majuscule (**MaClasse**)
→Les noms de **variables**, de **fonctions** et de **modules** doivent être de la forme : **ma_variable fonction_test_27() mon_module**, c'est-à-dire en minuscules avec un caractère « souligné » (« tiret du bas » ou *underscore* en anglais) pour séparer les différents « mots » dans le nom.
→Une **constante** s'écrit en majuscules (**MA_CONSTANTE)**

5 - Méthode destructeur

En POO, le destructeur d'une classe est une méthode spéciale lancée lors de la destruction d'un objet afin de récupérer les ressources (principalement la mémoire vive), réservée dynamiquement lors de l'instanciation de l'objet. En Python, le constructeur porte le nom **__del__**. Ce dernier n'est autant nécessaire que dans autres langages car Python dispose d'un ramasse-miettes (garbage collector) qui gère automatiquement la gestion de la mémoire. Cette méthode peut être appelée

explicitement ou automatiquement par le garbage collector lorsque toutes les références à l'objet ont été supprimées, c'est-à-dire lorsqu'un objet est ramassé.

```
 8 class Eleve:
 9     '''Un élève est caractérisé par:
10     -->ine: Identifiant national eleve
11     --> nom
12     --> et prénom'''
13
14     #Constructeur
15     def __init__(self, ine, nom, prenom):
16         self.ine = ine
17         self.nom = nom
18         self.prenom = prenom
19         print("Création réussie d'objet")
20
21         #destructeur
22     def __del__(self):
23         print("Destruction de l'objet")
24
25
26 eleve = Eleve("A89786754BC", "DUPONT", "Hugo")
27 print("INE:{0}, NOM:{1}, Prénom:{2}".format(eleve.ine,eleve.nom,eleve.prenom ))
28 del eleve
```

Dans cet exemple, nous avons utilisé le mot-clé **del** (ligne 28) qui va supprimer toutes les références de l'objet 'eleve', donc le destructeur est appelé automatiquement.

Notons que le destructeur est appelé également après la fin du programme (lorsqu'on a zéro références)

6 - Attributs de classe et d'instance

→ Un **attribut d'instance** (propriété d'instance) est associé à une instance de la classe et non à la classe. Chaque objet possède donc sa propre copie de la propriété. Par exemple, si vous créez plusieurs objets de type **Eleve**, les attributs *ine*, *nom*, *prenom* de chacun ne seront pas forcément identiques d'un objet à l'autre.

Eleve instance

ine	"A89786754BC"
nom	"DUPONT"
prenom	"Hugo"

Eleve instance

ine	"A89786456BD"
nom	"PASSONI"
prenom	"Clara"

```
eleve_1 = Eleve("A89786754BC", "DUPONT", "Hugo")
eleve_2 = Eleve("A89786456BD", "PASSONI", "Clara")
print("INE de l'élève 1:", eleve_1.ine)
print("NOM de l'élève 2:", eleve_2.nom)
```

Affichage obtenu :

```
Création réussie d'objet
Création réussie d'objet
INE de l'élève 1: A89786754BC
NOM de l'élève 2: PASSONI
```

Pour **appeler un attribut d'instance, il faut le préfixer par le nom de l'objet**, par exemple : **objet.attribut (eleve_1.ine).**

Il est à noter que si on veut instancier un objet de type **Eleve**, il faut fournir les trois attributs d'instance : *ine*, *nom* et *prenom*, sinon, on obtient une erreur avec cette instruction :

eleve_0 = Eleve()

TypeError: __init__() missing 3 required positional arguments: 'ine', 'nom', and 'prenom '

→ Un **attribut de classe** (propriété de classe) est associé à sa classe et non à une instance de cette classe. Cet attribut est utile lorsque les objets doivent partager des données identiques. On le déclare directement dans le corps de la classe avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe (**MaClasse.attribut_classe**). Et on y accède de cette façon également, en dehors de la classe.

```
 8 class Eleve:
 9     '''Un élève est caractérisé par:
10     -->ine: Identifiant national eleve
11     --> nom
12     --> et prénom'''
13
14     nombre_eleves = 0
15
16     #Constructeur
17     def __init__(self, ine, nom, prenom):
18         self.ine = ine
19         self.nom = nom
20         self.prenom = prenom
21         print("Création réussie d'objet N°:",  Eleve.nombre_eleves+1)
22         Eleve.nombre_eleves += 1
23
24         #destructeur
25     def __del__(self):
26         print("Destruction de l'objet")
27
28
29 eleve_1 = Eleve("A89786754BC", "DUPONT", "Hugo")
30 eleve_2 = Eleve("A89786456BD", "PASSONI", "Clara")
31 print("Nombre d'élèves inscrits: {0} élève(s).".format(Eleve.nombre_eleves))
```

Analysons ce code :

→ Ligne 14 : on a déclaré un attribut de classe : **nombre_eleves**.

→ Ligne 22 : À chaque fois qu'on crée un objet de type **Eleve**, l'attribut de classe **nombe_eleves** s'incrémente de 1 (on aura au total, 2 élèves créés)

→ Ligne 31 : pour appeler l'attribut de classe **nombre_eleves**, on le préfixe par le nom de la classe :

Eleve.nombre_eleves

On obtient l'affichage suivant :

```
Création réussie d'objet N°: 1
Création réussie d'objet N°: 2
Nombre d'élèves inscrits: 2 élève(s).
```

7 - Méthodes d'instance :

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions agissant sur l'objet ou comportements de ce dernier. Créer une méthode d'instance, revient à créer une fonction ayant comme premier paramètre le mot clé **self (toujours)**.

Une méthode d'instance est donc associée à une instance d'une classe et non à la classe. Chaque objet possède donc sa propre copie de la méthode. Sa syntaxe est la suivante :

def nom_de_la_methode(….) :

Il n'y a pas de tirets bas avant et après le nom d'une méthode ordinaire.

```
 8 class Eleve:
 9     '''Un élève est caractérisé par:
10     -->ine: Identifiant national eleve
11     --> nom
12     --> et prénom'''
13
14     nombre_eleves = 0
15
16     #Constructeur
17     def __init__(self, ine, nom, prenom):
18         self.ine = ine
19         self.nom = nom
20         self.prenom = prenom
21         print("Création réussie d'objet N°:",  Eleve.nombre_eleves+1)
22         Eleve.nombre_eleves += 1
23
24         #destructeur
25     def __del__(self):
26         print("Destruction de l'objet")
27
28     def mon_nom_prenom(self):
29         print("Je m'appelle {0} {1}".format(self.prenom, self.nom))
30
31 eleve_1 = Eleve("A89786754BC", "DUPONT", "Hugo")
32 eleve_2 = Eleve("A89786456BD", "PASSONI", "Clara")
33 eleve_1.mon_nom_prenom()
34 eleve_2.mon_nom_prenom()
35 print("Nombre d'élèves inscrits: {0} élève(s).".format(Eleve.nombre_eleves))
```

Analysons ce code :

→ Lignes 28 et 29 contiennent la définition de la méthode : **mon_nom_prenom(self)**. Elle permet d'afficher le nom et le prénom d'un élève.

→ Lignes 33 et 34 représentent des appels à la méthode (**nom_objet.nom_de_la_methode()**), par exemple : eleve_2.mon_nom_prenom().

On obtient l'affichage suivant :

```
Création réussie d'objet N°: 1
Création réussie d'objet N°: 2
Je m'appelle Hugo DUPONT
Je m'appelle Clara PASSONI
Nombre d'élèves inscrits: 2 élève(s).
```

8 - Méthode de représentation :

Soit l'instance suivante : eleve_1 = Eleve("A89786754BC", "DUPONT", "Hugo")

Lorsque on exécute l'instruction suivante : **print(eleve_1)**, on obtient ce résultat d'affichage :

<__main__.Eleve object at 0x7f0a8c907ef0>

On obtient son adresse mémoire et son type object, mais on affiche pas ses caractéristiques. Pour pallier à ce problème d'affichage, nous allons utiliser une méthode spéciale dite de représentation ou d'affichage : **def __str__(self) :**

Dans cette méthode, nous allons convertir notre objet en une chaîne de caractères :

```
def __str__(self):
    return "{0} {1}, INE: {2}".format(self.prenom, self.nom, self.ine)
```

Pour afficher une instance, il suffit de faire : **print(eleve_1).** On obtient l'affichage suivant :

Hugo DUPONT, INE: A89786754BC

Il y a une autre possibilité pour la méthode d'affichage : on peut écrire la méthode **__repr__**. Ainsi, on peut afficher une instance sans utiliser print().

```
def __repr__(self):
    return "{0} {1}, INE: {2}".format(self.prenom, self.nom, self.ine)
```

En tapant le nom de l'instance dans la console, on obtient :

```
In [28]: eleve_1
Out[28]: Hugo DUPONT, INE: A89786754BC
```

Attention : si on travaille dans l'éditeur, il faut impérativement utiliser print dans les deux cas pour déclencher un affichage dans la console.

III - Principes de la POO

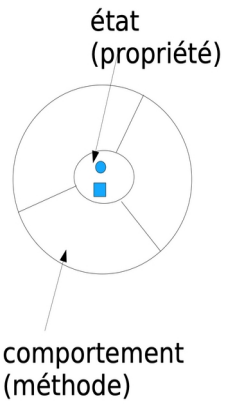
La POO est dirigée par trois fondamentaux qu'il convient de toujours garder à l'esprit : **encapsulation**, **héritage** et **polymorphisme**.
Nous allons nous intéresser principalement au premier principe qui est **l'encapsulation**.

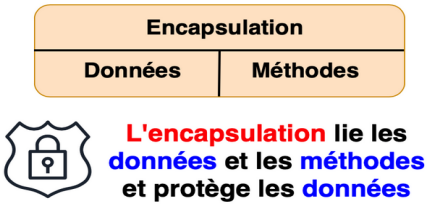
1 - Encapsulation

L'un des inconvénients de la programmation procédurale, c'est le manque de sécurité des données, c'est-à-dire, on déclare des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée.

En POO, L'**encapsulation** est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet (les informations liées à la structure de l'objet). Ainsi, un utilisateur n'ayant pas les droits requis, ne peut accéder aux données que par les interfaces (méthodes ou services définies pour modifier les données de l'objet). Cela permet d'assurer l'intégrité des données contenues dans l'objet.

L'encapsulation permet de définir le niveau de visibilité des éléments de la classe. Ils définissent les droits d'accès : **publique**, **privée**, **protégée.**





On va définir des méthodes un peu particulières, appelées des accesseurs (**getters**) et des mutateurs (**setters**). Les accesseurs donnent accès aux attributs. Les mutateurs permettent de les modifier. Concrètement, au lieu d'écrire **mon_objet.mon_attribut**, il faut écrire **mon_objet.get_mon_attribut()**. De la même manière, pour modifier l'attribut ce sera **mon_objet.set_mon_attribut(valeur)** et non pas **mon_objet.mon_attribut = valeur**.

Remarque :

Python n'implémente pas directement ces concepts de visibilité des membres de classe et il est donc impossible de définir un membre comme privé ou protégé : par défaut, tous les membres de classe sont publics en Python. En revanche, certaines conventions ont été mises en place par la communauté Python, notamment au niveau des noms des membres de classe qui servent à indiquer aux autres développeurs qu'on ne devrait accéder à tel membre que depuis l'intérieur de la classe ou à tel autre membre que depuis la classe ou une classe fille.

2 - Méthodes et attributs publics

Comme leur nom l'indique, les attributs et méthodes dits publics sont accessibles depuis tous les descendants et dans tous les modules. On peut considérer que les éléments publics n'ont pas de restriction particulière. Un attribut ne devrait être public que si sa modification n'entraîne pas de changement dans le comportement de l'objet. Dans le cas contraire, il faut passer par une méthode. Modifier un attribut "manuellement" et ensuite appeler une méthode pour informer de cette modification est une violation du principe d'encapsulation.

3 - Méthodes et attributs privés

Les membres privés (attributs et méthodes) sont des membres auxquels on ne peut accéder que depuis l'intérieur de la classe.

Par convention (car il n'y a aucun équivalent réel en Python), et pour informer d'autres développeurs du niveau de la visibilité souhaité pour un membre (attribut ou méthode), il suffit de suivre ces indications :

→ On préfixera les noms des membres qu'on souhaite définir comme "privés" avec deux underscores comme ceci : **__nom_du_membre** ;

→ On préfixera les noms des membres qu'on souhaite définir comme "protégés" avec un underscore comme ceci : **_nom_du_membre**.

Il est à noter que ces conventions n'ont pas été adoptées par hasard. En effet, Python possède un mécanisme appelé "**name mangling**" qui fait que tout membre de classe commençant par deux underscores, c'est-à-dire de la forme **__nom_du_membre** sera remplacé textuellement lors de l'interprétation par **_nom-de-classe__nom_du_membre**.

Cela fait que si un développeur essaie d'utiliser un membre défini avec deux underscores tel quel, Python renverra une erreur puisqu'il préfixera le nom avec un underscore et le nom de la classe du membre.

Exemple :

```
class Visibilite:
    '''Une classe avec trois attributs de visibilités différente'''

    var_publique = "Je suis une variable publique"
    _var_protegee = "Je suis une variable protégée"
    __var_privee = "Je suis une variable privee"
```

On définit ici une classe **Visibilite** qui contient trois variables : **var_publique**, **_var_protegee**, **__var_privee**. Notez qu'on pourrait aussi bien définir des fonctions de la même manière.

Maintenant, créons un objet à partir de cette classe et essayons d'afficher les valeurs de nos variables de classe à partir de l'objet créé :

```
In [30]: objet = Visibilite()

In [31]: objet.var_publique
Out[31]: 'Je suis une variable publique'

In [32]: objet._var_protegee
Out[32]: 'Je suis une variable protégée'

In [33]: objet.__var_privee
Traceback (most recent call last):

  File "<ipython-input-33-6b1e6555eacc>", line 1, in <module>
    objet.__var_privee

AttributeError: 'Visibilite' object has no attribute '__var_privee'


In [34]:

In [34]: print(dir(objet))
['_Visibilite__var_privee', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_var_protegee', 'var_publique']

In [35]: objet._Visibilite__var_privee
Out[35]: 'Je suis une variable privee'
```

Analysons cette trace d'exécution :

on accède sans souci à nos variables **var_publique** et **_var_protegee**. En revanche, pour **__var_privee**, le mécanisme de **name mangling** s'active et le nom passé est préfixé par un underscore et le nom de la classe (**_Visibilite__var_privee**). On peut cependant contourner cela et toujours afficher le contenu de la variable privée en utilisant la notation **_Visibiilite__var_privee**.

4 - Accesseurs et mutateurs

Pour manipuler les attributs d'instance, on utilisera de préférence des méthodes dédiées dont le rôle est de faire l'interface entre l'utilisateur de l'objet et la représentation interne de ce dernier (ses attributs). Il existe 2 familles :

→ les **accesseurs** (ou getters) : pour obtenir la valeur d'un attribut.

→ les **mutateurs** ou modificateurs (ou setters) : pour modifier la valeur d'un attribut.

On peut rendre certains attributs accessibles depuis l'extérieur de la classe ou l'inverse, ou les rendre non modifiables ou bien modifiables sous certaines conditions.

Un accesseur est une méthode qui commence par le mot **get**, par exemple : **def get_nom_attribut(self) :**

Un mutateur est une méthode qui commence par le mot **set**, par exemple : **set_nom_attribut(self, nouvelle_valeur) :**

Exemple : Dans le code ci-dessous, nous avons l'attribut __ine qui est privé. Nous avons rendu cet attribut accessible (avec la méthode **def get_ine(self)**). Pour la modification, on souhaite garder la main sur la façon d'attribuer un INE à un élève afin de ne pas avoir des incohérences (par exemple si un INE ne comporte pas 11 caractères, alors il est faux)

```
class Eleve:
    '''Un élève est caractérisé par:
    -->ine: Identifiant national eleve
    --> nom
    --> et prénom'''

    nombre_eleves = 0

    #Constructeur
    def __init__(self, ine, nom, prenom):
        self.__ine = ine
        self.nom = nom
        self.prenom = prenom
        print("Création réussie d'objet N°:",  Eleve.nombre_eleves+1)
        Eleve.nombre_eleves += 1
    #Accesseur
    def get_ine(self):
        return self.__ine

    #Modificateur
    def set_ine(self, nouveau_ine):
        '''Si le nouveau ine n'est pas composé de 11 caractères
        on affiche une erreur'''
        if len(nouveau_ine) == 11:
            self.__ine = nouveau_ine
        else:
            print("ERREUR!!! INE doit contenir 11 caractères")

eleve_1 = Eleve("A89786754BC", "DUPONT", "Hugo")
eleve_2 = Eleve("A89786456BD", "PASSONI", "Clara")
print(eleve_1.get_ine()) #appel accesseur
eleve_1.set_ine("AZ234")#appel modificateur
print(eleve_1.get_ine())#appel accesseur
```

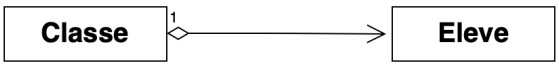
On obtient l'affichage suivant :

```
Création réussie d'objet N°: 1
Création réussie d'objet N°: 2
A89786754BC
ERREUR!!! INE doit contenir 11 caractères
A89786754BC
```

5 - Notion d'agrégation de classes

La conception d'une classe a pour but généralement de pouvoir créer des objets qui suivent tous le même modèle de fabrication.
Lorsque l'on souhaite modéliser une relation tout/partie entre deux classes, où une classe constitue un élément plus grand (tout) composé d'éléments plus petits (partie), il faut utiliser une **agrégation**. Une agrégation est une association qui représente une relation d'inclusion structurelle ou comportementale d'un élément dans un ensemble. Graphiquement, on ajoute un losange vide du côté de l'agrégat. Une agrégation n'entraîne pas de contrainte sur la durée de vie des parties par rapport au tout.

Prenons un exemple de classe et d'élève. Un seul élève ne peut pas appartenir à plusieurs classes, mais si nous supprimons la classe, l'objet élève sera *non* détruit. La classe joue le rôle d'ensemble :



Voici le script permettant de créer la classe **Classe** et d'implémenter la relation d'agrégation qui existe entre un élève et sa classe :

```
class Classe:
    def __init__(self, nom, liste_eleves = []):
        self.nom = nom
        self.liste_eleves = liste_eleves

    def ajouter_eleve(self, nouveau_eleve):
        self.liste_eleves.append(nouveau_eleve)


    def retirer_eleve(self, eleve):
        self.liste_eleves.remove(eleve)


    def __str__(self):
        if self.liste_eleves == []:
            return "La liste des élèves de la classe:" +self.nom + "est vide"
        else:
            chaine = "La liste des élèves de la classe:" +self.nom
            for indice, eleve in enumerate(self.liste_eleves):
                chaine += "\n"+ str(indice + 1) + ": "+ eleve.__str__()
        return chaine


#programme principale
classe_premiere = Classe("Première")
classe_premiere.ajouter_eleve(eleve_1)
classe_premiere.ajouter_eleve(eleve_2)
print(classe_premiere)
```

Voici un schéma récapitulatif de la trace d'exécution du script ci-dessus :

```
Création réussie d'objet N°: 1
Création réussie d'objet N°: 2
La liste des élèves de la classe:
1: Hugo DUPONT
```