PROGRAMMATION ORIENTEE OBJET

Le langage **Simula 67** pose les constructions qui seront celles des langages orientés objet, Mais c'est réellement avec **Smalltalk 71** puis **Smalltalk 80** que les principes de la programmation par objets, résultat des travaux d'**Alan Kay**, sont véhiculés.

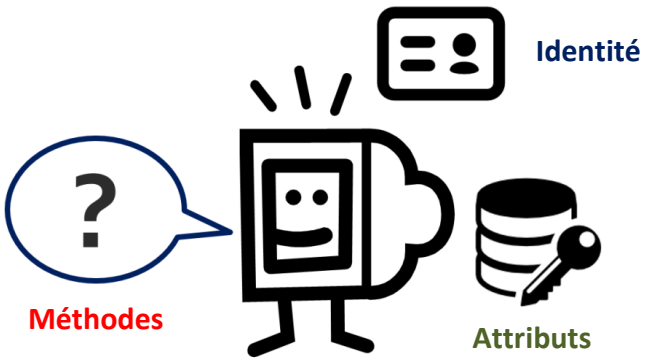
Les années 1980 voient l'apparition des premiers langages à objets : **C++**, **Objective-C**, **Eiffel**, **Common Lisp Object System**, etc. Les années 1990 voient l'âge d'or de l'extension de la programmation par objets dans les différents secteurs du développement logiciel notamment avec les langages **Python**, **PHP**, **Java**, **Ruby**, **C#**, **VB.NET**, etc.

1 – PROGRAMMATION ORIENTEE OBJET

1.1 – Notion d'objet

La **programmation orientée objet** (POO) consiste dans la définition et l'interaction de briques logicielles appelées **objets**. Un objet est une entité informatique représentant un sujet et contenant des informations et des outils manipulés dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel. Un objet est caractérisé par 3 notions :

- Une **identité** unique : L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. Cette identité est définie à la déclaration de l'objet (**instanciation**) en donnant un nom à l'objet.
- Des **attributs** (ou données) qui caractérisent l'objet. Ils s'agit de variables qui stockent des informations d'**état** de l'objet.
- Des **méthodes** (ou traitements) qui permettent de définir le **comportement de l'objet** c'est-à-dire l'ensemble des actions que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier



1.2 – Notion de classe : un premier exemple

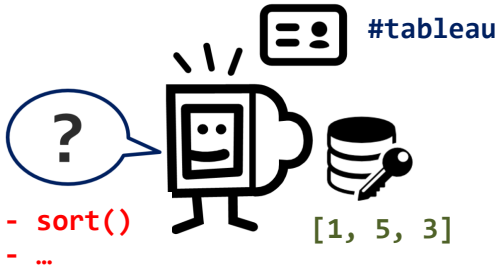
Un objet est créé à partir d'un modèle appelé **classe** ou **prototype**, dont il hérite les comportements et les caractéristiques.

Dans le langage Python, le type de données **`list`** est une classe.

```
>>> tableau = [1, 5, 3]
>>> type(tableau)
<class 'list'>
```

Une action possible sur les objets de type **`list`** est le tri de celle-ci avec la méthode nommée **`sort()`**.

```
>>> tableau = [1, 5, 3]
>>> tableau.sort()
>>> tableau
[1, 3, 5]
```



On peut afficher les méthodes associées à un objet avec la fonction **`dir(objet)`** :

```
>>> tableau = [1, 5, 3]
>>> dir(tableau)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

On retrouve les méthodes connues concernant type **`list`** : **`sort()`**, **`count()`**, **`append()`**, **`pop()`** … Les méthodes encadrées par les doubles underscores sont des **méthodes spéciales**.

Remarque : Dans le langage Python, tout est objet, les entiers, les flottants, même les modules. Par exemple le module **`math`** est un objet comprenant l'attribut **`pi`**, la méthode **`sqrt()`**, …

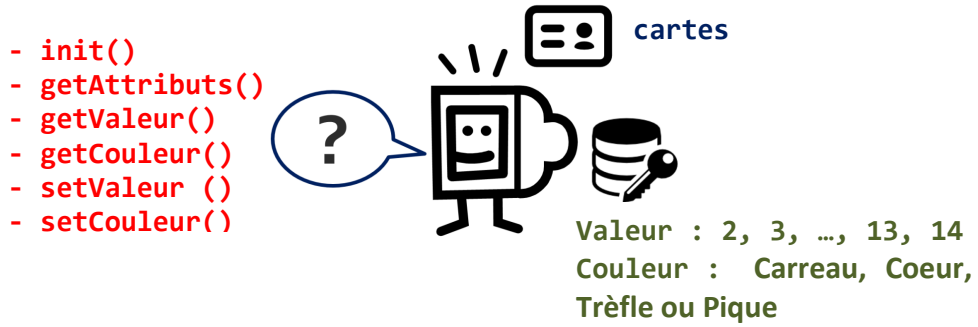
1.3 – Notion de classe : vocabulaire

- Le type de données avec ses **caractéristiques** et ses **actions** possibles s'appelle **classe**.
- Les **caractéristiques** (ou variables) de la classe s'appellent les **attributs**.
- Les **actions** possibles à effectuer avec la classe s'appellent les **méthodes**.
- La **classe** définit donc les **attributs** et les actions possibles sur ces attributs, les **méthodes**.
- On dit que les **attributs** et les **méthodes** sont **encapsulés** dans la **classe**.
- Un objet du type de la classe s'appelle une **instance** de la classe et la création d'un objet d'une classe s'appelle une **instanciation** de cette classe. Lorsqu'on **définit les attributs** d'un objet de la classe, on parle donc d'**instanciation**.

2 – CREATION D'UNE CLASSE

Pour illustrer le cours, nous allons créer une classe simple correspondant à une carte d'un jeu de **52 cartes**.
Les objets de cette classe possèderont 2 attributs :

- la **valeur** : 2, 3⋯ , 10, 11 pour le Valet, 12 pour la Dame, 13 pour le Roi et14 pour l'As;
- la **couleur** : Carreau, Coeur, Trèfle ou Pique.



Par convention, le nom d'une classe s'écrit toujours avec une majuscule :

```
class Carte:
    "Une carte d'un jeu de 52 cartes"
```

2.1 – Constructeur

Une classe possède toujours une méthode spéciale appelée **constructeur**. C'est la méthode qui est appelée lors la création d'un nouvel objet appartenant à la classe. Elle permet d'**initialiser les attributs**, c'est-à-dire l'état de l'objet.

En Python, le constructeur est la méthode **__init__** :

```
def __init__(self, parametres):
```

Le paramètre **self**, désigne l'objet auquel s'appliquera la méthode. Il représente l'objet dans la méthode en attendant qu'il soit créé.

```
class Carte:
    "Une carte d'un jeu de 52 cartes"
    def __init__(self, valeur, couleur):
        self.valeur = valeur
        self.couleur = couleur
```

2.2 – Création d'une instance de la classe

La création d'un objet de la classe **Carte**, son constructeur est appelé implicitement et l'ordinateur alloue de la mémoire pour l'objet et ses attributs. On peut d'ailleurs obtenir l'adresse mémoire de notre objet créé. Dans l'exemple ci-dessous, l'objet carte1 instance de la classe Carte a été crée.

```
>>> carte1 = Carte(9, "tréfle")
>>> carte1
<__main__.Carte object at 0x035299D0>
```

Il possible **d'accéder aux valeurs des attributs** d'un objet en utilisant l'opérateur d'accessibilité :

```
nom_objet.nom_attribut
```

```
>>> carte1.valeur
9
```

```
>>> carte1.couleur
'tréfle'
```

Il également possible **de modifier la valeur des attributs** d'un objet :

```
>>> carte1.valeur = carte1.valeur + 1
>>> carte1.valeur
10
```

2.3 – Visibilité et encapsulation

Cependant, il est souvent conseillé de cacher la représentation interne des attributs. C'est-à-dire qu'un objet doit protéger ses données. On parle alors d'**encapsulation**.

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**, c'est-à-dire en empêchant l'accès aux données par un autre moyens que les méthodes dédiées qui jouent le rôle d'**interface**. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

L'encapsulation permet de définir le type de **visibilité** des attributs de la classe. Il existe deux grand types de visibilité :

- **publique** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privés** (private) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe ou bien par des **méthodes dédiées**.

Pour rendre privé, un attribut ou une méthode, il faut un double underscore « __ » devant son nom :

```
class Carte:
    "Une carte d'un jeu de 52 cartes"
    def __init__(self, valeur, couleur):
        self.__valeur = valeur
        self.__couleur = couleur
```

```
>>> carte1 = Carte(9, "tréfle")
>>> carte1.valeur
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
AttributeError: 'Carte' object has no attribute 'valeur'
```

2.4 – Les accesseurs ou « getters », les mutateurs ou « setters »

Pour accéder à la valeur d'un attribut nous utiliserons la **méthode des accesseurs** (ou « getters »). Le préfixe « **get** » (récupérer) est généralement utilisé pour nommer les accesseurs.

```
class Carte:
    "Une carte d'un jeu de 52 cartes"
    def __init__(self, valeur, couleur):
        self.__valeur = valeur
        self.__couleur = couleur
    def getAttributs(self):
        return (self.__valeur, self.__couleur)
```

```
>>> carte1 = Carte(9, "tréfle")
>>> carte2 = Carte(14, "pique")
>>> carte1.getAttributs()
(9, 'tréfle')
```

```
>>> carte2.getAttributs()
(14, 'pique')
```

Question 1 : Créer deux autres méthodes permettant de récupérer la valeur de la carte et la couleur avec les "getters" (accesseurs) **`getCouleur()`** et **`getValeur()`**.Tester le script.

Pour modifier les valeurs des attributs, il faut utiliser des méthodes particulières appelées **mutateurs** (ou « **setters** ») qui vont modifier la valeur d'un attribut d'un objet. Le préfixe « **`set`** » est généralement utilisé pour nommer les mutateurs.

```
class Carte:
    "Une carte d'un jeu de 52 cartes"
    def __init__(self, valeur, couleur):
        self.__valeur = valeur
        self.__couleur = couleur
    def getAttributs(self):
        return (self.__valeur, self.__couleur)
    def getCouleur(self):
        return self.__couleur
    def getValeur(self):
        return self.__valeur
    def setValeur(self, val):
        if 2<= val <=14:
            self.__valeur = val
```

```
>>> carte1 = Carte(9, "tréfle")
>>> carte1.getAttributs()
(9, 'tréfle')
```

```
>>> carte1.setValeur(2)
>>> carte1.getAttributs()
(2, 'tréfle')
```

Question 2 :

1. Créer le mutateur de l'attribut « **`couleur`** ».
2. Créer une instance « **`carte2`** » qui a pour valeur « **`roi`** » et couleur de « **`cœur`** » puis modifier sa valeur en la passant à « **`dame`** ».
3. Modifier la couleur de l'objet « **`carte2`** » en le passant à « **`pique`** ».
4. Modifier la « **`carte2`** » en le passant à 8 de carreau.

2.5 – Les méthodes particulières en Python

Il existe en Python un certain nombre de méthodes particulières possédant chacune un nom spécifique entouré d'un **double underscore** « __ » comme le constructeur **__init__**. Parmi ces différentes méthodes on trouve entre autres :

- **__str__(self)** qui renvoie une chaine de caractère définie dans la méthode.
- **__lt__(self, autre_objet)** qui permet de faire une comparaison entre deux objets.
- **__len__(self)** qui renvoie la taille de l'objet.

```
class Carte:
    "Une carte d'un jeu de 52 cartes"
    def __init__(self, valeur, couleur):
        self.__valeur = valeur
        self.__couleur = couleur

    def __str__(self):
        return "Votre carte est le "+str(self.__valeur)+" de "+ self.__couleur

    def __lt__(self, c):
        return self.__valeur < c.__valeur
```

```
>>>print(Carte(10, 'Coeur'))
```

```
Votre carte est le 10 de Coeur
```

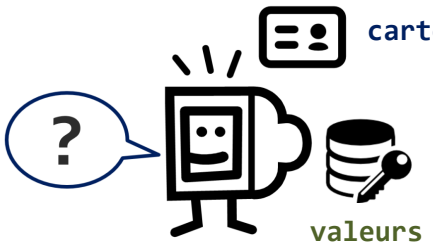
```
>>> Carte(10, 'Coeur') < Carte(8, 'Trefle')
```

```
False
```

3 – EXERCICE

Soit la classe « **Carte** », ci-dessous, dont on vous donne les en-têtes de méthodes. Cette classe diffère légèrement de la classe « **Carte** » vue plus haut.

- init()
- getNom()
- getCouleur()
- getValeur()
- setNom()
- setCouleur()
- egalite
- estSuperieurA
- estInferieurA



valeurs : '2':2, '3':3, …, 'Roi':13, 'As':14
noms : '2', '3', …, 'Roi', 'As'
couleurs : Carreau, Coeur, Trefle, Pique

```
class Carte:
    def __init__(self, nom, couleur):
        # Affectation de l'attribut nom et de l'attribut couleur
        couleurs = ('Carreau', 'Coeur', 'Trefle', 'Pique')
        noms = ['2', '3', '4', '5', '6', '7', '8', '9', '10',
                'Valet', 'Dame', 'Roi', 'As']
        valeurs = {'2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8,
                   '9':9 , '10':10, 'Valet':11, 'Dame':12, 'Roi':13, 'As':14}

    def setNom(self, nom):
    # Mutateur de l'attribut nom (de la liste noms)

    def getNom(self):
    # renvoie le nom de la carte (de la liste noms): Accesseur

    def getCouleur(self):
    # renvoie la couleur de la carte (de la liste couleur): Accesseur

    def getValeur(self):
    # renvoie la valeur de la carte (du dictionnaire valeurs) : Accesseur

    def egalite(self, carte):
        ''' Renvoie True si les cartes self et carte ont même valeur, False sinon
            carte: Objet de type Carte
        '''

    def estSuperieureA(self, carte):
        ''' Renvoie True si la valeur de self est supérieure à celle de carte,
            False sinon
            carte: Objet de type Carte
        '''

    def estInferieureA(self, carte):
        ''' Renvoie True si la valeur de self est inférieure à celle de carte,
        False sinon
        carte: Objet de type Carte
        '''
```

Question 3 :

1. Compléter le constructeur et les différentes méthodes de la classe « **`Carte`** ».
2. Editer le programme principal permettant de :
   - Créer la carte Valet de Coeur que l'on nommera c1.
   - Afficher le nom, la valeur et la couleur de c1.
   - Créer la carte As de Pique que l'on nommera c2.
   - Afficher le nom, la valeur et la couleur de c2.
   - Modifier le nom de la carte c2 en Roi et afficher le nom, la valeur et la couleur de c2.
   - Créer la carte 8 de Trefle que l'on nommera c3.
   - Comparer les cartes c1 et c2 puis c1 et c3