

PROGRAMMATION ORIENTEE OBJET

Classes, Objets, Attributs

Jusqu'à présent, les programmes que vous avez réalisés suivaient un raisonnement classique, ou procédural. Chaque programme était décomposé en fonctions élémentaires réalisant des tâches relativement simples.

Lorsque plusieurs programmeurs travaillent simultanément sur un même projet, il est nécessaire de programmer autrement afin d'éviter les conflits entre les fonctions. Cette programmation correspond en quelque sorte à des échanges de messages par des entités de base appelées « **Objet** ». C'est ce que l'on appelle la **POO** : la **P**rogrammation **O**rientée **O**bjets.

Par exemple, on peut trouver dans la bibliothèque Tkinter, une classe **Button()** à partir de laquelle on peut créer dans une fenêtre un nombre quelconque de boutons.

Nous allons à présent examiner comment vous pouvez vous-mêmes définir de nouvelles classes d'objets. Il s'agit là d'un sujet relativement ardu, mais vous l'aborderez de manière très progressive, en commençant par définir des classes d'objets très simples, que vous perfectionnerez ensuite.

Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui soient elles-mêmes des objets, ceux-ci étant faits à leur tour d'autres objets plus simples, etc.

1. Utilité des classes

Les **classes** sont les principaux outils de la **P**rogrammation **O**rientée **O**bjets ou **POO** (**O**bject **O**riented **P**rogramming ou **OOP**). Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.

Le premier bénéfice de cette approche de la programmation consiste dans le fait que **les différents objets utilisés peuvent être construits indépendamment les uns des autres** (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence.

Ce résultat est obtenu grâce au concept d'**encapsulation** : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « **enfermés** » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des **procédures** bien définies.

L'utilisation de classes dans vos programmes vous permettra - entre autres choses **d'éviter au maximum l'emploi de variables globales**. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, surtout dans les programmes volumineux, parce qu'il est toujours possible que de telles variables soient modifiées ou même redéfinies n'importe où dans le corps du programme (et ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel).

Un second bénéfice résultant de l'utilisation des **classes** est la possibilité qu'elles offrent de **construire de nouveaux objets à partir d'objets préexistants**, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de **dérivation** et de **polymorphisme**.

- La **dérivation** est le mécanisme qui permet de construire une classe « **enfant** » au départ d'une classe « **parente** ». L'enfant ainsi obtenu **hérite** toutes les propriétés et toute la fonctionnalité de son ancêtre, auxquelles on peut ajouter ce que l'on veut.
- Le **polymorphisme** permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte.

La programmation orientée objet est optionnelle sous Python. Vous pouvez donc mener à bien de nombreux projets sans l'utiliser, avec des outils plus simples tels que les fonctions. Sachez cependant que les classes constituent des outils pratiques et puissants. Une bonne compréhension des classes vous aidera notamment à maîtriser le domaine des interfaces graphiques (Tkinter, wxPython), et vous préparera efficacement à aborder d'autres langages modernes tels que C++ ou Java.

2. Définition d'une classe élémentaire

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction « **class** ». Nous allons donc apprendre à utiliser cette instruction, en commençant par définir un type d'**objet** très rudimentaire, lequel sera simplement un nouveau type de donnée. Nous avons déjà utilisé différents types de données, mais c'étaient à chaque fois des types de données intégrées dans le langage lui-même.

Nous allons maintenant créer un nouveau type composite : le type « **Point** ».

Ce type correspondra au concept de « **point** » en Mathématiques.

Dans un espace à deux dimensions, un point est caractérisé par deux nombres (ses coordonnées suivant x et y). En notation mathématique, on représente donc un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point (25,17). Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type « float ». Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet.

Pour y arriver, nous allons définir une classe **Point()** :

```
class Point(object):  
    "Définition d'un point mathématique"
```

Les définitions de classes peuvent être situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer). L'exemple ci-dessus est probablement le plus simple qui se puisse concevoir. Une seule ligne nous a suffi pour définir le nouveau type d'objet **Point()**.

D'après Apprendre à programmer avec Python de Gérard Swinnen :

Remarquons d'emblée que :

- L'instruction **class** est un nouvel exemple d'**instruction composée**. N'oubliez pas le **double point** « : » obligatoire à la fin de la ligne, et l'**indentation** du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne. Dans notre exemple ultra-simplifié, cette ligne n'est rien d'autre qu'un **simple commentaire**. Par convention, si la première ligne suivant l'instruction **class** est une chaîne de caractères, celle-ci sera considérée comme un commentaire (associé à l'attribut prédéfini « **__doc__** »), et incorporée automatiquement dans un dispositif de documentation des classes qui fait partie intégrante de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit.

- Les parenthèses sont destinées à contenir la référence d'une classe préexistante. Cela est requis pour permettre le mécanisme d'**héritage**. Toute classe nouvelle que nous créons peut en effet hériter d'une classe parente un ensemble de caractéristiques, auxquelles elle ajoutera les siennes qui lui sont propres. Lorsque l'on désire créer une classe fondamentale, c'est-à-dire ne dérivant d'aucune autre, comme c'est le cas ici avec notre classe **Point()**, la référence à indiquer doit être par convention le nom spécial « **object** », lequel désigne l'ancêtre de toutes les classes.

- Rappelez-vous aussi la convention qui consiste à toujours donner aux classes **des noms qui commencent par une majuscule**. Dans la suite de ce texte, nous respecterons encore une autre convention qui consiste à associer à chaque nom de classe une paire de parenthèses, comme nous le faisons déjà pour les noms de fonctions.

Nous venons de définir une classe « **Point()** ».

Nous pouvons dès à présent nous en servir pour créer des « **objets** » de ce type (appelés aussi « instances »), par **instanciation**. Créons par exemple un nouvel objet **p9** :

```
p9 = Point()
```

Remarque : Sous Python, on peut donc instancier un objet à l'aide d'une simple instruction d'affectation. D'autres langages imposent l'emploi d'une instruction spéciale, souvent appelée « **new** » pour bien montrer que l'on crée un nouvel objet à partir d'un moule.

Exemple : **p9 = new Point()**

Après cette instruction, la variable « **p9** » contient la référence d'un nouvel objet « **Point()** ». Nous pouvons dire également que « **p9** » est une nouvelle « **instance** » de la classe « **Point()** ».

Attention : Comme les fonctions, les classes auxquelles on fait appel dans une instruction doivent toujours être accompagnées de parenthèses (même si aucun argument n'est transmis). Nous verrons un peu plus loin que les classes peuvent être appelées avec des arguments.

Remarquez bien cependant que la définition d'une classe ne nécessite pas de parenthèses (contrairement à ce qui est de règle lors de la définition des fonctions), sauf si nous souhaitons que la classe en cours de définition dérive d'une autre classe préexistante.

Nous continuerons cependant à les utiliser nous-mêmes dans la suite de ce texte, afin de bien marquer l'importance du concept d'héritage.

Voyons maintenant si nous pouvons faire quelque chose avec notre nouvel objet **p9** :

Testez cette commande dans la console.

```
*** Remote Interpreter Reinitialized ***
>>> print(p9)
<__main__.Point object at 0x000001A796D0A610>
>>> |
```

Le message renvoyé par Python indique, comme vous l'aurez certainement bien compris tout de suite, que « **p9** est une instance de la classe « **Point()** », laquelle est définie elle-même au niveau principal (**main**) du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale.

Testez cette commande dans la console.

```
*** Remote Interpreter Reinitialized ***
>>> print(p9.__doc__)
Définition d'un point mathématique
>>> |
```

Les documentations de divers objets Python sont associées à l'attribut prédéfini « **__doc__** ». Il est donc toujours possible de retrouver la documentation associée à un objet Python quelconque, en invoquant cet attribut.

3. Attributs (ou variables) d'instance

L'objet que nous venons de créer est juste une « coquille » vide. Nous allons à présent lui ajouter des composants, par simple assignation, en utilisant le système de qualification des noms par points. (« . »).

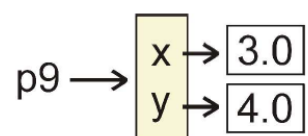
*Ce système de notation est similaire à celui que nous utilisons pour désigner les variables d'un module, comme par exemple « **math.pi** ». Nous aurons l'occasion d'y revenir plus tard, mais sachez dès à présent que les modules peuvent en effet contenir des fonctions, mais aussi des classes et des variables.*

```
p9.x = 3.0
p9.y = 4.0
```

Les variables **x** et **y** que nous avons ainsi définies en les liant d'emblée à **p9**, sont désormais des **attributs** de l'objet **p9**. On peut également les appeler des **variables d'instance**. Elles sont en effet incorporées, ou plutôt **encapsulées** dans cette instance (ou objet).

Le diagramme d'état ci-contre montre le résultat de ces affectations :

La variable **p9** contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs **x** et **y**. Ceux-ci contiennent les références des valeurs 3.0 et 4.0, mémorisées ailleurs.



On pourra utiliser les attributs d'un objet dans n'importe quelle expression, exactement comme toutes les variables ordinaires.

Testez ces commandes dans la console.

```
*** Remote Interpreter Reinitialized ***
>>> print(p9.x)
3.0
>>> print(p9.x**2 + p9.y**2)
25.0
>>> |
```

Du fait de leur **encapsulation** dans l'objet, les **attributs** sont des variables distinctes d'autres variables qui pourraient porter le même nom. Par exemple, l'instruction « **x = p9.x** » signifie : «> extraire de l'objet référencé par **p9** la valeur de son attribut **x**, et assigner cette valeur à la variable **x** ». Il n'y a pas de conflit entre la variable indépendante **x**, et l'attribut **x** de l'objet **p9**.

L'objet **p9** contient en effet son propre espace de noms, indépendant de l'espace de nom principal où se trouve la variable **x**.

Important : les exemples donnés ici sont provisoires.

Nous venons de voir qu'il est très aisé d'ajouter un attribut à un objet en utilisant une simple instruction d'assignation telle que « **p9.x = 3.0** ». On peut se permettre cela sous Python (c'est une conséquence de son caractère foncièrement dynamique), mais cela n'est pas vraiment recommandable, comme vous le comprendrez plus loin. Nous n'utiliserons donc cette façon de faire que de manière anecdotique, et uniquement dans le but de simplifier nos premières explications concernant les attributs d'instances. La bonne manière de procéder sera développée dans le chapitre suivant.

4. Passage d'objets comme arguments dans l'appel d'une fonction

Les fonctions peuvent utiliser des objets comme paramètres, et elles peuvent également fournir un objet comme valeur de retour. Par exemple, vous pouvez définir une fonction telle que celle-ci :

```
def affiche_point(p):
    print("coord. horizontale =", p.x, "coord. verticale =", p.y)
```

Le paramètre « **p** » utilisé par cette fonction doit être un **objet** de type « **Point()** », dont l'instruction qui suit utilisera les variables d'instance « **p.x** » et « **p.y** ». Lorsqu'on appelle cette fonction, il faut donc lui fournir un objet de type **Point()** comme argument. Essayons avec l'objet **p9** :

Testez cette commande dans la console.

```
*** Remote Interpreter Reinitialized ***
>>> affiche_point(p9)
coord. horizontale = 3.0 coord. verticale = 4.0
>>> |
```

Exercice-1 :

Ajouter une fonction « **distance(p1, p2)** » qui permette de calculer la distance entre deux points (p1 et p2), en utilisant la classe **Point()** créée dans le début de ce cours.

```
from math import sqrt

class Point(object):
    "Définition d'un point mathématique"

def affiche_point(p):
    print("coord. horizontale =", p.x, "coord. verticale =", p.y)
```

Cette fonction attendra évidemment deux objets **Point()** comme arguments.

Vous prendrez :

```
p5.x, p5.y, p6.x, p6.y = 12.3, 5.7, 6.2, 9.1
```

Tester alors les fonctions

- affiche_point(p) pour chaque point p5 et p6,
- distance(p1,p2)

```
*** Remote Interpreter Reinitialized ***
>>> affiche_point(p5)
coord. horizontale = 12.3 coord. verticale = 5.7
>>> affiche_point(p6)
coord. horizontale = 6.2 coord. verticale = 9.1
>>> print("La distance entre les points p5 et p6 est égale à :", distance(p5,p6))
La distance entre les points p5 et p6 est égale à : 6.983552104767315
>>>
```

5. Similitude et unicité

Dans la langue parlée, les mêmes mots peuvent avoir des significations fort différentes suivant le contexte dans lequel on les utilise. La conséquence en est que certaines expressions utilisant ces mots peuvent être comprises de plusieurs manières différentes (expressions ambiguës).

Le mot « même », par exemple, a des significations différentes dans les phrases :
« Charles et moi avons la même voiture » et « Charles et moi avons la même mère »
Dans la première, ce que je veux dire est que la voiture de Charles et la mienne sont du même modèle. Il s'agit pourtant de deux voitures distinctes. Dans la seconde, j'indique que la mère de Charles et la mienne constituent en fait une seule et unique personne.

Lorsque nous traitons d'objets logiciels, nous pouvons rencontrer la même ambiguïté.

Par exemple, si nous parlons de l'égalité de deux objets **Point()**, cela signifie-t-il que ces deux objets contiennent les mêmes données (leurs attributs), ou bien cela signifie-t-il que nous parlons de deux références à un même et unique objet ? Considérez par exemple les instructions suivantes :

```
p1 = Point()
p1.x = 3
p1.y = 4
p2 = Point()
p2.x = 3
p2.y = 4
```

Ces instructions créent deux objets **p1** et **p2** qui restent distincts, même s'ils font partie d'une même classe et ont des contenus similaires.

```
*** Remote Interpreter Reinitialized ***
>>> print(p1==p2)
False
```

Cette instruction teste l'égalité de ces deux objets (double signe égal), et le résultat est « **False** » : il n'y a donc pas égalité.

On peut confirmer cela d'une autre manière encore :

```
*** Remote Interpreter Reinitialized ***
>>> print(p1)
<__main__.Point object at 0x000001F3F5E60BE0>
>>> print(p2)
<__main__.Point object at 0x000001F3F5E60D00>
>>> |
```

L'information est claire : les deux variables « **p1** » et « **p2** » référencent bien des objets différents, mémorisés à des emplacements différents dans la mémoire vive de l'ordinateur.

Ajoutez cette ligne de code.

```
p1=p2
```

Par l'instruction « **p2 = p1** », nous assignons le contenu de **p1** à **p2**. Cela signifie que désormais ces deux variables **réfèrent le même objet**. Les variables **p1** et **p2** sont des **alias** l'une de l'autre.

Puis tester à nouveau :

```
*** Remote Interpreter Reinitialized ***
>>> print(p1==p2)
True
```


Le test d'égalité dans l'instruction suivante renvoie cette fois la valeur « **True** », ce qui signifie que l'expression entre parenthèses est vraie : **p1** et **p2** désignent bien toutes deux un seul et unique objet, comme on peut s'en convaincre en essayant encore avec :

```
p1.x=7
```

On affecte la valeur **7** à l'attribut « **x** » de l'objet « **p1** » de la classe « **Point()** ».

```
*** Remote Interpreter Reinitialized ***
>>> print(p2.x)
7
>>>
```

Lorsqu'on modifie l'attribut « **x** » de l'objet « **p1** », on constate que l'attribut « **x** » de « **p2** » a changé, lui aussi.

Vérifions que les deux instances « **p1** » et « **p2** » de la classe « **Point()** » pointent vers le même emplacement de la mémoire vive.

```
*** Remote Interpreter Reinitialized ***
>>> print(p1)
<__main__.Point object at 0x00000189FB5C0D00>
>>> print(p2)
<__main__.Point object at 0x00000189FB5C0D00>
>>> |
```

6. Objets composés d'objets

Supposons maintenant que nous voulions définir une classe qui servira à représenter des **rectangles**.

Pour simplifier, nous allons considérer que ces rectangles seront toujours orientés horizontalement ou verticalement.

De quelles informations avons-nous besoin pour définir de tels rectangles ?

Il existe plusieurs possibilités. Nous pourrions par exemple spécifier la position du centre du rectangle (deux coordonnées) et préciser sa taille (largeur et hauteur).

Nous pourrions aussi spécifier les positions du coin supérieur gauche et du coin inférieur droit.

Ou encore la **position du coin supérieur gauche** et la **taille**. Admettons ce soit cette dernière convention qui soit retenue.

Définissons donc notre nouvelle classe « **Rectangle()** »:

```
class Rectangle(object):  
    "definition d'une classe de rectangles"
```

Créons alors l'instance (ou objet) « **boite** » de la classe « **Rectangle()** ». Puis associons-lui les attributs « **largeur** » et « **hauteur** ».

```
boite = Rectangle()  
boite.largeur = 50.0  
boite.hauteur = 35.0
```

Pour spécifier le coin supérieur gauche, nous allons créer une nouvelle **instance** (objet) de la classe « **Point()** », instance que l'on appelle « **coin** » (La classe « **Point()** » est la classe définie plus haut dans ce cours).

Nous lui associons ensuite les **attributs** « **x** » et « **y** ».

```
boite.coin = Point()  
boite.coin.x = 12.0  
boite.coin.y = 27.0
```

Ainsi nous avons créé un **objet**, à l'intérieur d'un autre **objet** !

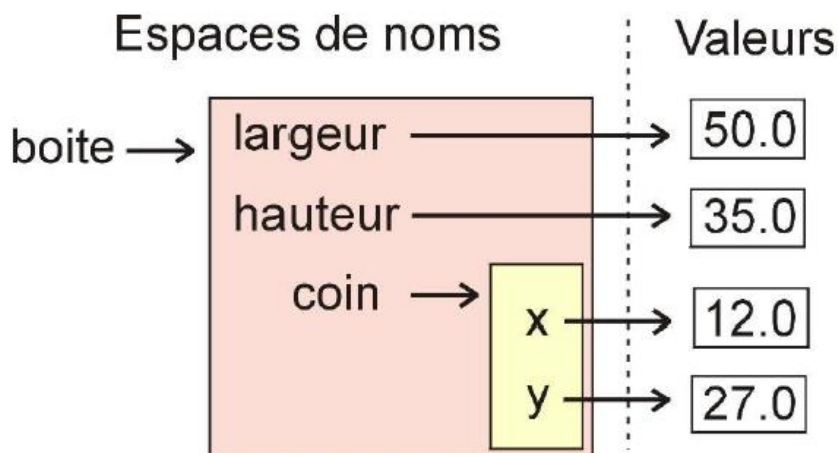
A la première de ces trois instructions, nous créons un nouvel attribut (objet) « **coin** » pour l'objet « **boite** » de la classe « **Point()** ».

Pour accéder à cet objet qui se trouve lui-même à l'intérieur d'un autre objet, nous utilisons la **qualification des noms hiérarchisée** (à l'aide de points « . »).

Ainsi l'expression « **boite.coin.y** » signifie :

- Aller à l'objet référencé dans la variable « **boite** ».
- Dans cet objet, repérer l'attribut « **coin** »,
- Puis aller à l'objet référencé dans cet attribut.
- Une fois cet autre objet trouvé, sélectionner son attribut « **y** ».

Vous pourrez peut-être mieux vous représenter tout cela à l'aide d'un diagramme tel que celui-ci :



Le nom « **boite** » se trouve dans l'**espace de noms principal**. Il référence un autre **espace de noms** réservé à l'objet correspondant, dans lequel sont mémorisés les noms **largeur**, **hauteur** et **coin**.

Ceux-ci référencent à leur tour, soit **d'autres espaces de noms** (cas du nom « **coin** »), soit **des valeurs** bien déterminées, lesquelles sont mémorisées ailleurs.

Python réserve des espaces de noms différents pour chaque **module**, chaque **classe**, chaque **instance**, chaque **fonction**. Vous pouvez tirer parti de tous ces espaces de noms bien compartimentés afin de réaliser des **programmes robustes**, c'est-à-dire des programmes dont les différents composants ne peuvent pas facilement interférer.

7. Objets comme valeurs de retour d'une fonction

Nous avons vu plus haut que les fonctions peuvent utiliser des objets comme paramètres. Elles peuvent également transmettre une instance comme valeur de retour. Par exemple, la fonction « **trouveCentre()** » ci-dessous doit être appelée avec un argument de type « **Rectangle()** » et elle renvoie un objet de type « **Point()** », lequel contiendra les coordonnées du centre du rectangle.

```
def trouveCentre(box):  
    p = Point()  
    p.x = box.coin.x + box.largeur/2.0  
    p.y = box.coin.y + box.hauteur/2.0  
    return p
```

Vous pouvez par exemple appeler cette fonction, en utilisant comme argument l'objet « **boite** » défini plus haut.

Testez ces commandes et commentez les résultats obtenus :

- trouveCentre(boite)
- centre = trouveCentre(boite)
- print(centre)
- print(centre.x, centre.y)
- affiche_point(centre)

```
*** Remote Interpreter Reinitialized ***  
>>> trouveCentre(boite)  
<__main__.Point object at 0x0000023E90F21FD0>  
>>> centre=trouveCentre(boite)  
>>> print(centre)  
<__main__.Point object at 0x0000023E90E59E50>  
>>> print(centre.x, centre.y)  
37.0 44.5  
>>> affiche_point(centre)  
coord. horizontale = 37.0 coord. verticale = 44.5
```

8. Modification des objets

Nous pouvons changer les propriétés d'un objet en assignant de nouvelles valeurs à ses attributs.

Par exemple, nous pouvons modifier la taille d'un rectangle (sans modifier sa position), en réassignant ses attributs hauteur et largeur.

Faites afficher les coordonnées actuelles du centre en utilisant la fonction « `affiche_point(p)` »

```
*** Remote Interpreter Reinitialized ***
>>> affiche_point(centre)
coord. horizontale = 37.0 coord. verticale = 44.5
>>> |
```

Ajouter ces lignes de code pour modifier la taille du rectangle.

```
boite.largeur = boite.largeur + 20
boite.hauteur = boite.hauteur - 5

centre=trouveCentre(boite)
```

Faites afficher les nouvelles coordonnées du point supérieur gauche.

```
*** Remote Interpreter Reinitialized ***
>>> affiche_point(centre)
coord. horizontale = 47.0 coord. verticale = 42.0
>>> |
```

Nous pouvons faire cela sous Python, parce que dans ce langage les propriétés des objets sont toujours **publiques**. D'autres langages établissent une distinction nette entre attributs publics (accessibles de l'extérieur de l'objet) et attributs privés (qui sont accessibles seulement aux algorithmes inclus dans l'objet lui-même).

Cependant, comme nous l'avons déjà signalé plus haut, modifier de cette façon les attributs d'une instance **n'est pas une pratique recommandable**, parce qu'elle contredit l'un des objectifs fondamentaux de la **Programmation Orientée Objet**, qui vise à établir une séparation stricte entre la fonctionnalité d'un objet (telle qu'elle a été déclarée au monde extérieur) et la manière dont cette fonctionnalité est réellement implémentée dans l'objet (et que le monde extérieur n'a pas à connaître).

Concrètement, cela signifie que nous devons maintenant étudier comment faire fonctionner les objets à l'aide d'outils vraiment appropriés, que nous appellerons des **méthodes**, et que nous allons étudier dans le chapitre suivant.

L'ensemble de ces méthodes constituera ce que nous appellerons désormais l'**interface** de l'objet.