ENSAM Bordeaux – mars 2020

Calcul Scientifique avec

python™ 3

- ▷ Introduction - Prise en main
- ▷ Les bases du langage Python
- ▷ I/O fichiers ASCII, tracés de courbes
- ▷ Les modules Numpy et Scipy
- ▶ Conception/programmation OO

Jean-Luc Charles - Éric Ducasse

Arts & Métiers, I2M

Arts et Métiers Sciences et Technologies

CC BY SA

I2M Bordeaux

" S'il n'y a pas de solution, c'est qu'il n'y a pas de problème."
– Logique Shadock

Les grandes étapes dans la conduite d'un projet (informatique ou pas) :

- **Analyse** : "QUOI faire ?"
  investigation du problème posé et des besoins

- **Conception** : "COMMENT faire ?"
  identifier les concepts de la solution

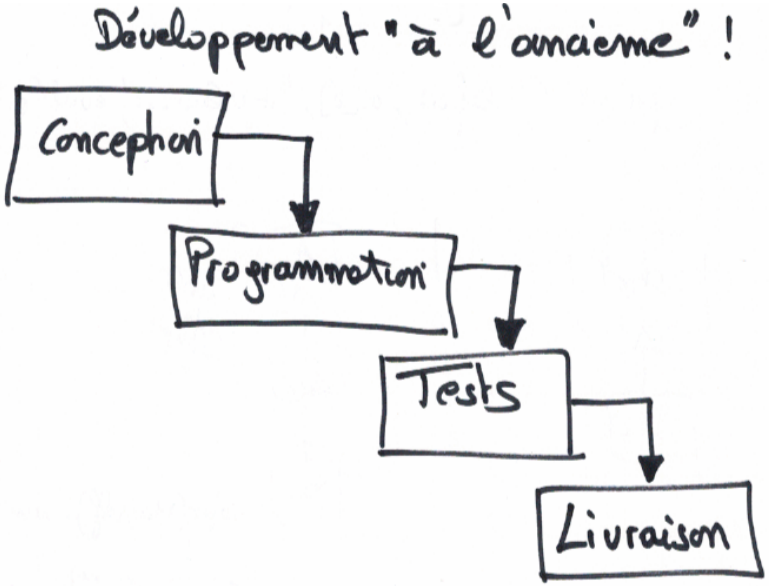
- **Réalisation** : "réaliser" la solution
  en informatique : programmer !

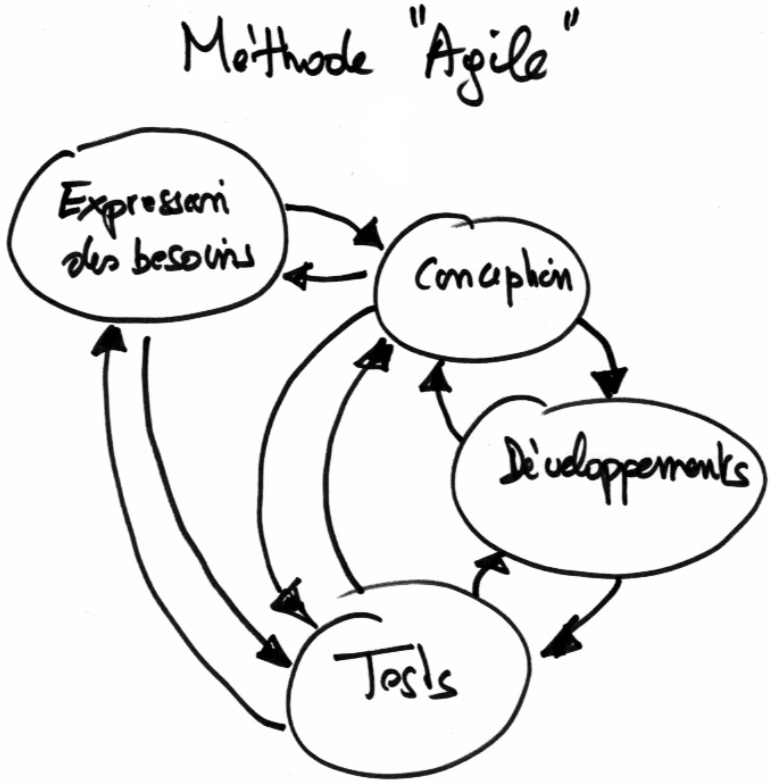
Plusieurs approches possibles dans la conduite d'un projet :

- **Approche Fonctionnelle** et les méthodes associées
  - Analyse Fonctionnelle (AF), SADT (Analyse Fonctionnelle descendante)
  - APTE (APplication aux Techniques d'Entreprise)
  - FAST (*Function Analysis System Technique*) ...
- **Approche Orientée Objet** et les méthodes associées
  - RUP (*Rational Unified Process*), XP (*Extreme Programming*) ...
  - Techniques de conception (*design pattern*, programmation par contrat, ...)

- De plus en plus utilisé : **Méthode Agile** qui enchaîne des cycles courts

conception $\rightsquigarrow$ programmation $\rightsquigarrow$ test





Approche Fonctionnelle (années 1960-2000)

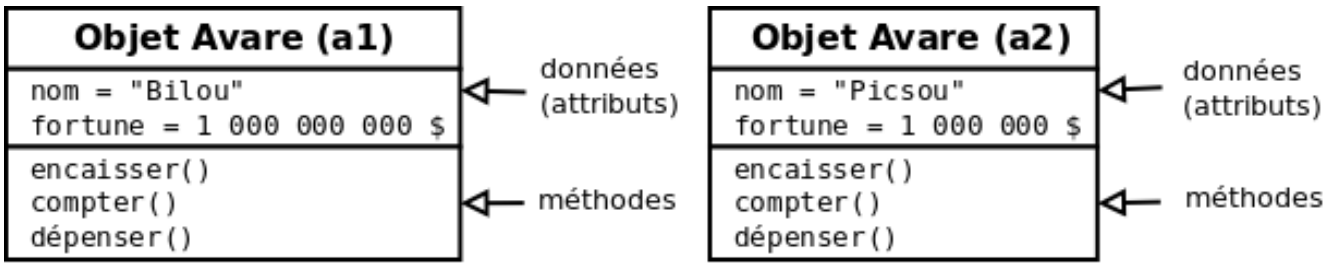
- Concepts clefs :
  - Les **fonctions** du système étudié (objet matériel, programme, service, ...)
    - identification des fonctions...
    - découpages en sous-fonctions...
  - **Analyse Fonctionnelle** : Fonctions Principales, Fonctions Contraintes ...

- Inconvénients :
  - Modification donnée ⇒ impacts multiples
    (toutes les fonctions utilisant la donnée)
  - Évolutivité et maintenabilité parfois problématique

Donne de "bons résultats" pour des problèmes aux fonctions bien identifiées, stables dans le temps (cahier des charges fixé, ne bouge pas).

Approche Orientée Objet (années 1980->...)

- Idée de l'approche objet
  S'inspirer du monde réel ... fait d'**objets**, **communiquant** entre eux, possédant :
  - des **propriétés**, intrinsèques ou dépendant d'éléments extérieurs
  - des **comportements**, intrinsèques ou dépendant d'éléments extérieurs.
- Un **objet** = { des **données**, des **méthodes** travaillant sur les données }
  - les **DONNÉES** : propriétés, attributs, données membres...
  - les **MÉTHODES** : fonctions membres, manipulant les données (définissent le comportement).



Deux objets de type Avare.

Analyse/Conception Orientée Objet

- Le comportement global du système étudié repose sur :
  - l'ensemble des objets identifiés,
  - les relations et les communications possibles entre ces objets.

  L'aspect dynamique du système correspond à des envois de messages entre objets qui déclenchent des traitements...

- Système étudié vu par l'approche Orientée Objet :

  Ensemble d'objets **interagissant entre eux** pour réaliser les **fonctions** du système.

- L'approche "Orienté Objet" est applicable pour l'Analyse, la Conception des systèmes et la Programmation !

Approche Orientée Objet

- Les paradigmes de l'Orienté Objet
  - **Objet**
  - **Classe**
  - **Encapsulation**
  - **Interface**
  - **Héritage**
  - **Polymorphisme**
  - **Généricité** (non traitée avec Python).

- Les langages OO (C++, C#, Java, VB.NET, Objective C, Ruby, Ada, PHP, Smaltalk, Python, Eiffel...) supportent tout ou partie de ces paradigmes...

Le concept d'objet : définition

- L'objet est **unité cohérente** possédant :
  - une **identité** ;
  - un **état**, défini par des données (**attributs**, ou **données membres**)
  - un **comportement**, défini par des fonctions (**méthodes**, ou **fonctions membres**).

- Un objet peut représenter :
  - un **objet concret** du monde réel, ayant une réalité physique
    une personne, une voiture, un outil, un système mécanique…
  - un **concept abstrait**
    un compte bancaire, une contrainte mécanique, une vitesse…
  - une **activité** produisant des effets observables
    un calcul numérique, un pilote d'impression…

Le principe d'encapsulation

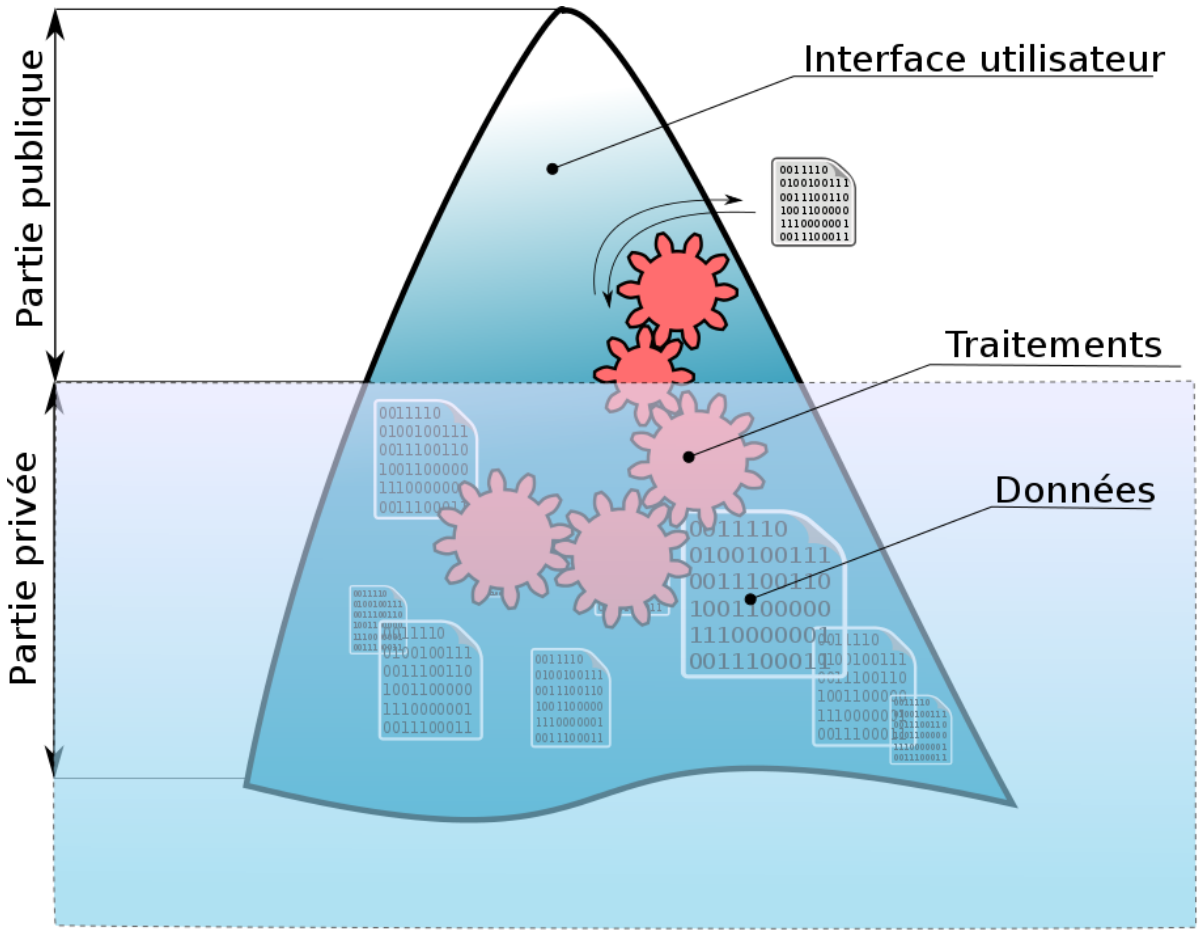
permet aux objets de se présenter sous deux vues possibles :

- **la vue externe** (celle de l'*utilisateur de l'objet*) : **comment on utilise** l'objet
  - Définit l'**interface** de l'objet
  - Fournit les services accessibles aux utilisateurs de l'objet
  - Ne fournit aucun accès aux mécanismes internes de l'objet
  - UML : déclarations qualifiées de **publiques** (*public*).

- **la vue interne** (celle du *concepteur* de l'objet) : **comment est construit** l'objet
  - Détaille la constitution interne de l'objet
  - UML : déclarations qualifiées de **privées** (*private*) .

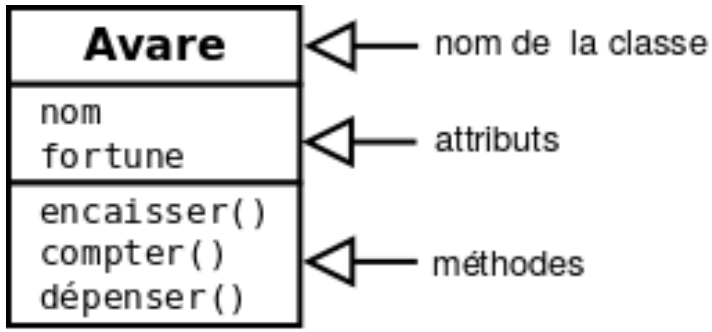
Le principe d'encapsulation

La métaphore de l'iceberg...



Le concept de classe

- **modèle** repésentant une famille d'objets :
  - même **structure** de données (*c.a.d.* même liste d'attributs)
  - mêmes **méthodes**
  - mêmes **relations**.



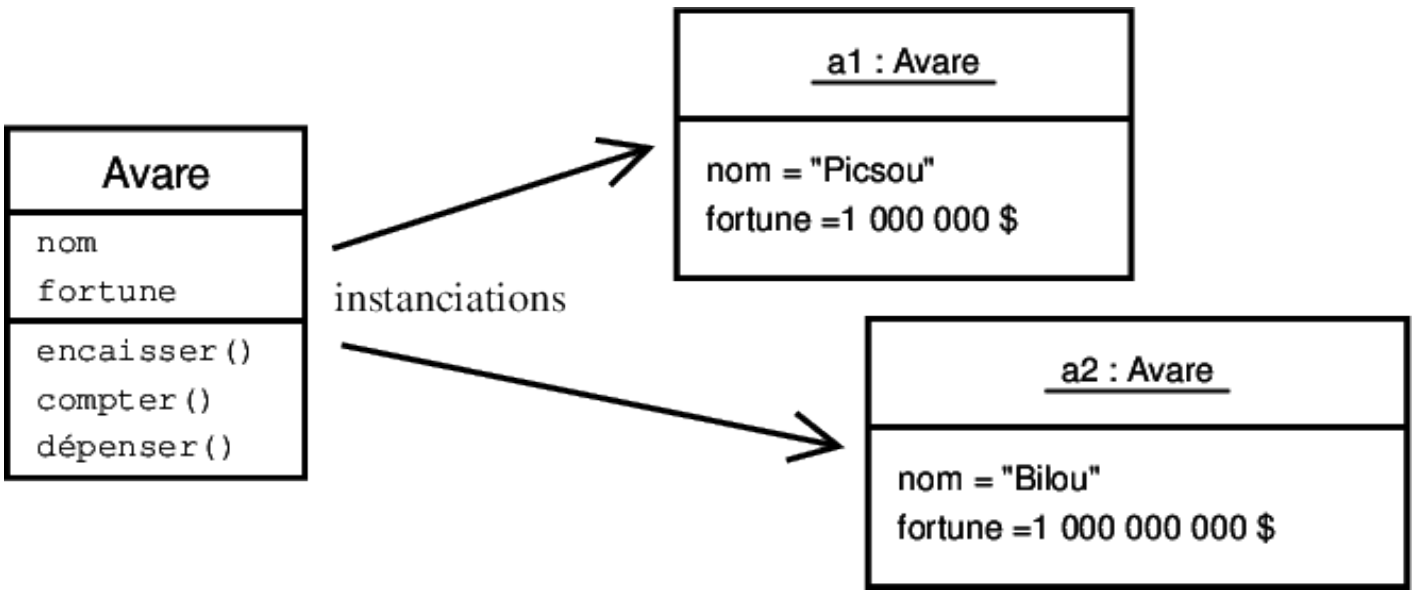
- La **classe** est fortement liée à la notion de **type** des langages informatiques.
  - un nom de classe devient un nouveau type du langage
  - les opérateurs du langage (`*`, `+`, `/`...) peuvent être re-définis pour s'appliquer aux nouveaux types définis par les classes utilisateur.

La classe par elle-même **ne contient pas les valeurs des données** : c'est un **modèle** qui décrit la structure des données.

Objet = Instance d'une classe

- La création d'un **objet** en tant qu'exemplaire concret (contenant des données) d'une classe s'appelle une **INSTANCIATION**.

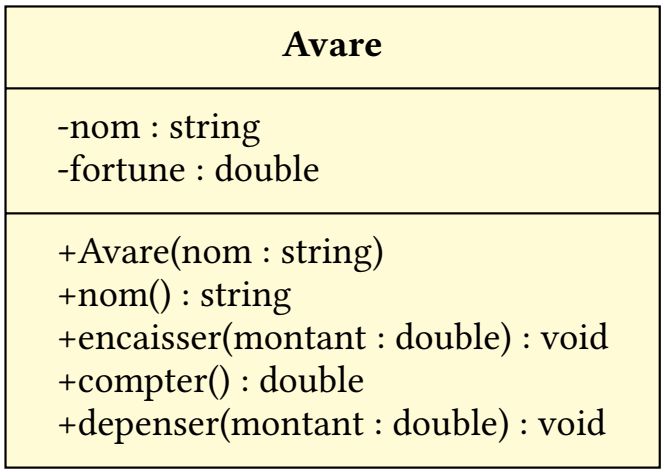
- Chaque **objet** (instance d'une classe) donne des **valeurs** aux **attributs** de la classe.



Le concept de classe : l'encapsulation

- La déclaration d'une classe permet de préciser complètement son **interface** :
  - la partie **publique**, utilisable par n'importe quelle entité
  - la partie **privée**, utilisable uniquement au sein de la classe
  - la partie **héritée**, utilisable par la classe ou des classe dérivées.

- Notation **UML** :
  - `-` attribut ou méthode privé (**private**)
  - `#` attribut ou méthode protégé (**protected**)
  - `+` attribut ou méthode publique (**public**).



Le concept de classe : Intérêts de l'encapsulation

- Facilite l'**évolution** du modèle :
  - on peut modifier des détails internes d'une classe (nombre, nature et noms des attributs, détails des traitements) sans modifier son interface
  - la façon de s'en servir ne change pas pour les utilsateurs (modèles, programmes, ...) de la classe.

- Favorise l'**intégrité des données** :
  - permet d'interdire l'accès direct aux détails internes
  - permet de **contrôler** et **garantir** la **cohérence des données** de la classe en imposant l'utilisation de méthodes publiques d'accès aux données de la classe (lecture ou écriture).

Le concept de classe

Exemple de **déclaration** de la classe Avare en langage C++ :

```
class Avare
{
public:
  // tout ce qui est public est accessible à tout le monde.
  Avare(const string & nom);    // Le constructeur : même nom que la classe
  const string & nom();
  void encaisser(double montant);
  void depenser(double montant);
  double compter() const;

protected:
  // tout ce qui est protégé est transmis aux classes dérivées

private:
  // tout ce qui est privé est interne à la classe
  double fortune;
  string nom;
}
```

Le concept de classe

Exemple de **définition** de la classe Avare en langage Python :

[Avare.py]

```
class Avare:

    def __init__(self, nom):      # Le constructeur en Python : __init__
        self.__nom = nom          # self.__xx : attribut d'instance privé 'xx'
        self.__fortune = 100000   # why not !!!

    def nom(self):
        return self.__nom

    def encaisser(self, montant):
        self.__fortune += montant

    def depenser(self, montant):
        x = 0.9*montant             # objet (variable) temporaire
        print("je ne peux dépenser que {}".format(x))
        self.__fortune -= x         # avare !

    def compter(self):
        return self.__fortune
```

Relations entre classes

- L'analyse des systèmes (réels ou conceptuels) se fait souvent selon :
  - la **Classification** (approche hiérarchique)
  - la **Composition/Décomposition** (approche structurelle).

"un pommier *est un* arbre *composé* d'un tronc, de branches, de feuilles et de fleurs"
"un élément triangle T6 *est un* élément fini *composé* de 6 noeuds"

- L'approche OO propose plusieurs types de relations entre classes :
  - l'**Héritage**, traduit souvent une classification hiérarchique
  - l'**Agrégation/Composition**, traduit qu'un contenant contient des agrégats/-composants (décomposition structurelle)
  - l'**Association** simple, permet à 2 classes de se connaître
  - l'**Utilisation/Dépendance**, traduit le fait qu'une classe se sert d'une autre.

- Les **relations** entre classes modélisent les relations (les **liens**) entre objets.

Relations entre classes : Héritage

- Traduit la relation **"est un"**

  ```
  "Un roman est un livre"
  "Une voiture est un véhicule"
  "Une matrice symétrique est une matrice"
  ```

- Tout ce que la classe de base sait faire, la classe dérivée sait le faire "mieux" ou "différement".

- C'est **LE** mécanisme de transmission des propriétés (attributs et méthodes) d'une classe à une autre.

- C'est **LE** mécanisme fondamental pour faire évoluer un modèle (un programme).

- La **classe de base** (classe mère, parente) transmet toutes ses propriétés **transmissibles** (publiques ou protégées) aux **classes dérivées** (classe fille, enfant).

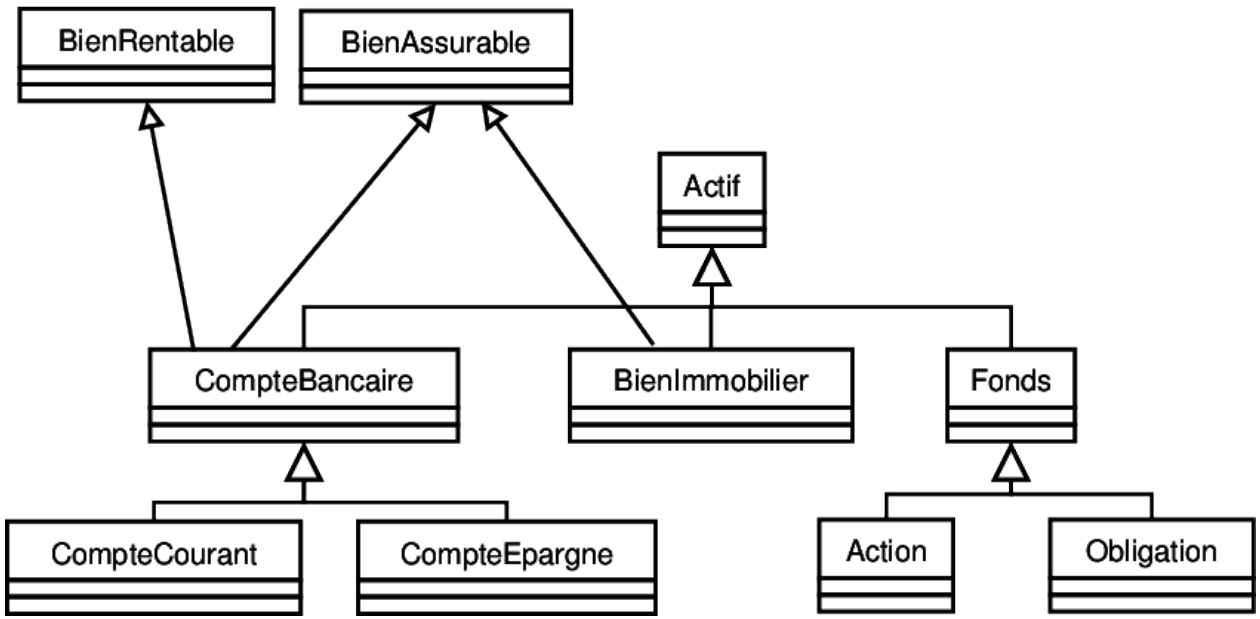
Relations entre classes : Héritage

- **simple** au **multiple** : une classe peut dériver d'une ou plusieurs classes de base.
- Souvent **arborescent** : une classe dérivée peut à son tour être une classe de base pour une autre dérivation (**hiérarchie de classes**).
- N'est pas réflexif : une classe ne peut pas hériter d'elle-même.
- La classe dérivée :
  - ne peut pas accéder aux membres privés de la classe de base
  - possède ses propres attributs/méthodes, que la classe de base ne connaît pas
  - peut redéfinir (améliorer, spécialiser....) les méthodes héritées de la classe de base.

L'**Héritage** est **LE** mécanisme fondamental pour faire évoluer un modèle, un programme.

Relations entre classes : Héritage

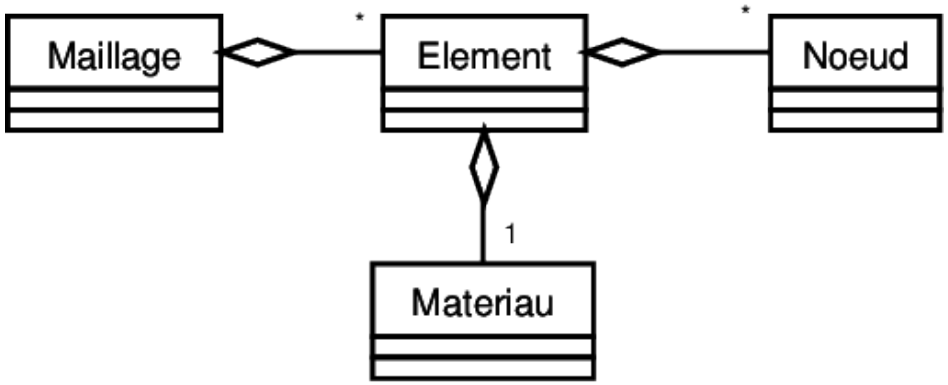
Exemple d'héritage simple et multiple :



Un CompteBancaire "*est un*" BienRentable,
mais "*est aussi un*" BienAssurable et "*un*" Actif.

Relations entre classes : Agrégation

- Traduit la relation "possède", "est constitué de", ou "est fait de".
- Relation bi-directionnelle dissymétrique exprimant un couplage fort :
  - traduit des relations de type **maître-esclave**, ou **contenant-contenu**, ...
  - l'une des classes est l'agrégat (le tout) composé de l'agrégation des parties.

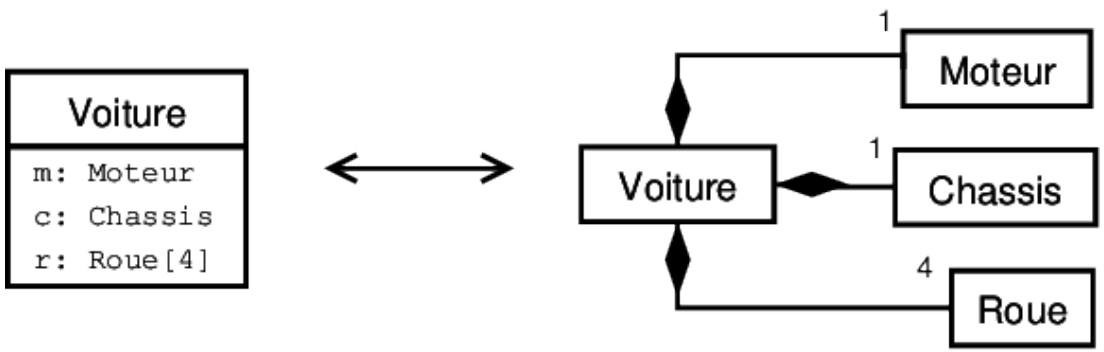


Un maillage éléments finis "est constitué" d'éléments.
Chaque élément "est fait" d'un matériau et "possède" des noeuds

Relations entre classes : Composition

- Forme d'agrégation avec un couplage encore plus fort
  - Les composants ne sont pas partageables entre plusieurs agrégats
  - La destruction de l'agrégat entraîne la destruction des composants agrégés
  - La composition agit comme un conteneur exclusif.

Les attributs d'une classe peuvent être vus comme des composants, au sens de la relation de composition :



Les premières approches objet

- L'émergence des **approches 'objet'** (1990-1995) :
  - Prise de conscience de l'importance d'une approche spécifiquement objet : comment structurer un système sans centrer l'analyse uniquement sur les données ou uniquement sur les traitements (mais sur les deux) ?
  - Plus de 50 "méthodes" objet sont apparues durant cette période (Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE...) !

- 3 approches ont émergé du lot :
  - **OMT** (*Object Modelling Technique*), par James Rumbaugh, centre de R&D de General Electric
  - **OOD** (*Object Oriented Design*), par Grady Booch
  - **OOSE** (*Object Oriented Software Engineering*), par Ivar Jacobson, centre de développement d'Ericsson, en Suède.

Object Management Group www.omg.org

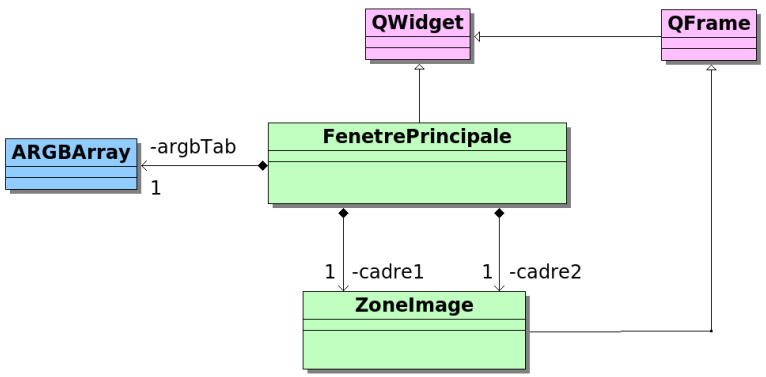
- Organisme à but non lucratif, créé en 1989 à l'initiative de grandes sociétés (HP, Sun, Unisys, American Airlines, Philips...)
- Unification et normalisation des approches objet.
- Fédère plus de 850 acteurs du monde informatique.
- Rôle : promouvoir des standards garantissant l'**interopérabilité** entre applications orientées objet.

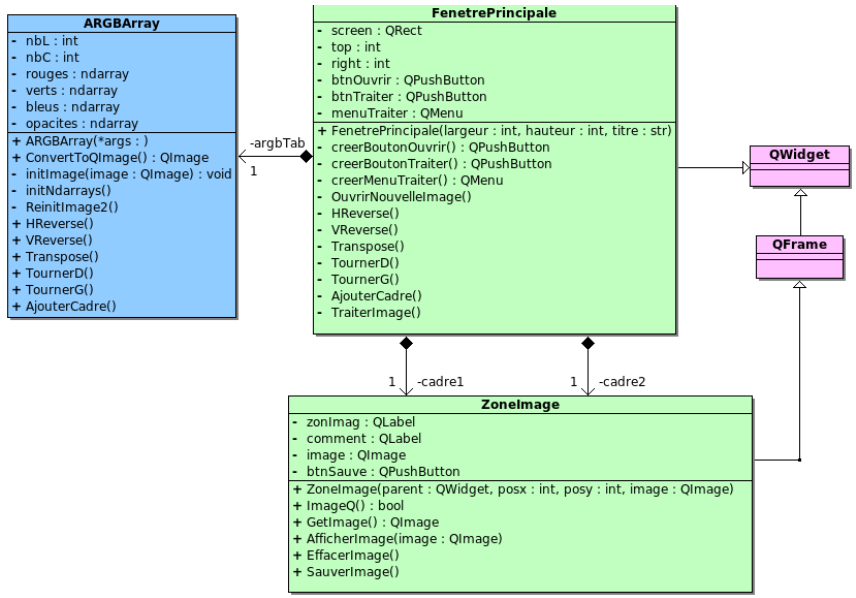
Unified Modelling Language www.uml.org

- Langage de modélisation **normalisé**, unifié, fusion des 3 méthodes : OMT, Booch et OOSE.
- Support de l'**approche objet** pour la **modélisation** des systèmes indépendamment de toute méthode ou langage de programmation.
- **Langage de modélisation objet** et non une **méthode**.
- Support graphique de communication.
- Cadre méthodologique.

Le language UML

- UML permet de réaliser des diagrammes (UML2 définit 9 diagrammes)
- Le principal diagramme est le **diagramme de classes** (global/détaillé...)





- De nombreux logiciels "diagrammeur UML" existent :
  fr.wikipedia.org/wiki/Comparaison_des_logiciels_d'UML
  uml.developpez.com
  - Par exemple : Modelio www.modelio.org, ArgoUML argouml.tigris.org, Bouml www.bouml.fr...

Définition d'une classe : mot clef `class`

```
class Point:                # le mot clef 'class' introduit la définition d'une classe

    def __init__(self):     # La méthode spéciale '__init__' joue le rôle du constructeur
        self.x = 0          # les attributs (publics) sont préfixés par self
        self.y = 0

p1 = Point()                # p1 : objet de type Point
print("1:",p1)
p1.x = 1; p1.y = 2;         # attributs publics accessibles en lecture/écriture !
print("2:",p1.x, p1.y)
p1.x = "n'importe quoi"     # danger ....
p1.z = 'ohoh';              # on peut même créer de nouveaux attributs (publics) !!!!
print("3:",p1.x, p1.y, p1.z)
print(dir(p1))              # voir tout ce qu'il y a dans l'objet p1
print(p1.__dict__)          # voir le dictionnaire des attributs de p1
```

```
1: <__main__.Point object at 0x7f6b46e9f748>
2: 1 2
3: n'importe quoi 2 ohoh
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'x', 'y', 'z']
{'z': 'ohoh', 'y': 2, 'x': 1}
```

Accès aux données d'un objet : syntaxe `objet.attribut`

```
class Point:                # le mot clef 'class' introduit la définition d'une classe
    def __init__(self):     # La méthode spéciale '__init__' joue le rôle du constructeur
        self.x = 0          # les attributs (publics) sont préfixés par 'self.'
        self.y = 0

        self.__z = 0        # les attributs "privés" sont préfixés par 'self.__'
                            # self.__z apparaît dans __dict__ comme '_Point__z' (mangling)

p1 = Point()                # p1 : objet de type Point
print(p1.__dict__)

p1.x = 1; p1.y = 2          # attributs publics accessibles en lecture/écriture !

# a = p1.z                  ->  AttributeError: 'Point' object has no attribute 'z'
# a = p1.__z                ->  AttributeError: 'Point' object has no attribute '__z'

p1.__z = 3                  # ERREUR logique ? : crée l'attribut public '__z' !!!
print(p1.__dict__)
```

```
{'_Point__z': 0, 'x': 0, 'y': 0}
{'_Point__z': 0, '__z': 3, 'x': 1, 'y': 2}
```

Accès aux méthodes d'un objet : syntaxe `objet.méthode(...)`

`classe.méthode(objet, args...)` est interprété comme `objet.méthode(args...)`

```
class Point:                # le mot clef 'class' introduit la définition d'une classe

    def __init__(self):     # La méthode spéciale '__init__' joue le rôle du constructeur
        self.x = 0          # les attributs sont préfixés par self
        self.y = 0

    def info(self):         # self est l'objet courant, obligatoire pour une méthode !
        print("x :", self.x, "y :", self.y)

p1 = Point()          # p1 : objet de type Point
p1.info()             # objet.méthode() : exécute 'méthode' avec les données de 'objet'.
print(dir(p1))        # voir tout ce qu'il y a dans l'objet p1
print(p1.__dict__) # les méthodes n'apparaissent pas dans le __dict__ de l'objet
print(Point.__dict__.keys()) # mais dans le __dict__ de la classe
```

```
x : 0 y : 0
['__class__'..., '__weakref__', 'info', 'x', 'y']
{'y': 0, 'x': 0}
dict_keys(['info', '__init__', '__weakref__', '__dict__', '__doc__', '__module__'])
```

Documentation des classes Python

Documentation avec les *docstrings* [class06.py]

```
class A:
    # En présence d'un docstring """...""" ou '''...''' les commmentaires avec des '#'
    # ne sont pas montrés par help(A), cf diapo page suivante...
    """ Exemple de 'Docstring' de la classe A, qui
    peut s'étendre sur plusieurs lignes"""

    def __init__(self):     # méthode spéciale "constructeur"
        """ le constructeur ne prend pas d'argument"""
        print("appel du constructeur de la classe A")
        self.y = 2          # attribut (public) y
        self.__z = 3        # attribut (privé) z

    def obtenir_z(self):
        """ permet de lire la valeur de l'attribut privé z"""
        return self.__z

    def changer_z(self, valeur):
        """ change sous contrôle la valeur de l'attribut privé z"""
        if 0 <= valeur <= 10:
            self.__z = valeur
        else:
            print("uniquement des valeurs entre 0 et 10 SVP !")
```

Documentation des classes Python

```
>>> help(A)
Help on class A in module __main__:

class A(builtins.object)
 |  Exemple de 'Docstring' de la classe A, qui
 |  peut s'étendre sur plusieurs lignes
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      le constructeur ne prend pas d'argument
 |
 |  changer_z(self, valeur)
 |      change sous contrôle la valeur de l'attribut privé z
 |
 |  obtenir_z(self)
 |      permet de lire la valeur de l'attribut privé z
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  ...
```

Redéfinition des opérateurs du langage

Un opérateur Python • écrit `a•b` est interprété comme `a.__∘__(b)`

- `__∘__` est la **fonction spéciale** associée à l'opérateur •
- `__∘__` est redéfinie dans les classes qui doivent supporter l'opérateur •.
- par exemple, pour les opérateurs numériques :
  docs.python.org/reference/datamodel.html#emulating-numeric-types

```
class Point:
    def __init__(self, x=0, y=0):
        self.__x = x      # les attributs privés sont préfixés par 'self.__'
        self.__y = y
    def __eq__(self, p):  # redéfinition de l'opérateur '=='
        return self.__x == p.__x and self.__y == p.__y

p1 = Point(1,1); p2 = Point(2,2); p3 = Point(1,1)
print("p1 == p2 :", p1 == p2); print("p1 == p3 :", p1 == p3)
```

```
p1   ==   p2   : False
p1   ==   p3   : True
```

Propriétés : accès en lecture

- Le décorateur `@property` permet de définir une **propriété** en lecture :
  - accès avec une *syntaxe d'attribut* `a = objet.name`
  - mécanisme interne : la valeur est renvoyée par l'exécution de la méthode associée.

```
from math import sqrt
class Point:
    def __init__(self, x=0, y=0):
        self.__x = x      # les attributs privés sont préfixés par 'self.__'
        self.__y = y
    @property             # Décorateur '@property'
    def distance(self):
        return sqrt(self.__x**2 + self.__y**2)

p1 = Point(1,1)                 # p1 : objet de type Point
print("distance:",p1.distance)  # exécution de la méthode associée
# p1.distance = 3               -> AttributeError: can't set attribute
```

```
distance: 1.4142135623730951
```

Propriétés : accès en écriture

- Le décorateur `@name.setter` permet de définir une **propriété** en écriture :
  - accès avec une syntaxe d'attribut `objet.name = a`
  - mécanisme interne : exécution de la méthode associée

```
from math import sqrt
class Point:
    def __init__(self, x=0, y=0):
        self.__x, self.__y = x, y  # attributs privés préfixés par 'self.__'
    @property
    def xvalue(self): return self.__x
    @xvalue.setter
    def xvalue(self, value): self.__x = value

p1 = Point(1,1)              # p1 : objet de type Point
print("xvalue:",p1.xvalue) # exécution de la méthode associée @property
p1.xvalue = 3                # exécution de la méthode associée par @xvalue.setter
print("xvalue:",p1.xvalue)
```

```
xvalue: 1
xvalue: 3
```

Héritage : accès aux données de la classe de base

```
class A():
    def __init__(self):      # Constructeur de la classe A :
        self.a = ['A']       #  définition de a, attribut public de A
        self.__x = -1        #  définition de x, attribut privé de A.

class B(A):
    def __init__(self):      # Constructeur de la classe B :
        super().__init__()   #  exécution du constructeur de la classe de base
        self.b = 0           #  définition de b : attribut public de B
        self.a.append('B')   #  utilisation de a : attribut public hérité de A
        # c = self.__x       -> ERREUR : x est privé dans la classe A !

class C(A):
    def __init__(self):    # Constructeur de la classe C :
        A.__init__(self)   #  exéc. constructeur de la classe A (autre syntaxe)
        self.a=2           #  définition de a : attribut public de C, masque A.a !

b = B(); print(b.__dict__)
c = C(); print(c.__dict__)
```

```
{'a': ['A', 'B'], '_A__x': -1, 'b': 0}
{'a': 2, '_A__x': -1}
```

Héritage : surcharge/accès aux méthodes de la classe de base

```
class A():
    def go(self): print("A is running go()")

class B(A):
    def go(self): print("B is running go()")

class C(B):
    def go(self):
        A.go(self); B.go(self)

class D(B): pass

if __name__ == "__main__":
    a = A(); print(" a.go() gives: ",end=""); a.go()
    b = B(); print(" b.go() gives: ",end=""); b.go()
    c = C(); print(" c.go() gives: ",end=""); c.go()
    d = D(); print(" d.go() gives: ",end=""); d.go()
```

```
 a.go() gives: A is running go()
 b.go() gives: B is running go()
 c.go() gives: A is running go()
B is running go()
 d.go() gives: B is running go()
```

Accès aux données d'un objet (avancé)

Comment l'interpréteur Python "interprète" l'accès aux données...

- `obj.name = value` est traduit par `obj.__setattr__("name", value)`
- `del obj.name` est traduit par `obj.__delattr__("name")`
- Comportement par défaut de `__setattr__` et `__getattr__` :
  - utiliser d'abord le dictionnaire `__dict__` des attributs de `obj` ;
  - si `name` est une **propriété** (*property*) : les opérations `set` et `delete` sont alors réalisées par celles des propriétés.
- Évaluer `obj.name` conduit à rechercher `obj.__getattribute__("name")`
- Comportement par défaut de `__getattribute__` :
  - rechercher `'name'` dans les **propriétés**, le `__dict__` local, puis les classes de base.
  - Si échec : la méthode `__getattr__` de la classe est utilisée, si elle est définie.

Héritage multiple

```
class A():
    pass

class B():
    pass

class D1(A,B):      # A et B sont 2 classes indépendantes, l'ordre indiffère
    pass

class D2(B,A):      # A et B sont 2 classes indépendantes, l'ordre indiffère
    pass
```

```
class A():
    pass

class B(A):         # B dérive de A
    pass

##class D1(A,B):  - > TypeError: Cannot create a consistent method resolution
##    pass

class D2(B,A):      # l'ordre est important car B dérive de A
    pass
```

Attributs de classe (statiques)

```
class Point:
    liste  = []      # définition de l'attribut de classe (statique) public 'liste'
    __nbPt = 0       # définition de l'attribut de classe (statique) privé 'nbpt'

    def __init__(self, x=0, y=0):
        self.__x = x       # définition d'un attribut d'instance -> préfixé par 'self.'
        self.__y = y
        Point.__nbPt += 1        # utilisation d'un attribut statiques : préfixer son
        Point.liste.append(self) # nom par le nom de la classe.

    def __str__(self):           # redéfinition (surcharge) de la fonction str()
        return "x: {0:f}, y: {1:f}".format(self.__x, self.__y)

Point(1,1); Point(2,2)  # création d'objets Point sans nom
p1=Point(3,3)
print(len(Point.liste)) # l'attribut statique 'liste' est public
print(Point.liste[0]); print(Point.liste[1])
# print(Point.__nbPt)   -> ERREUR : attribut statique 'nbPt' est privé !
print(len(p1.liste))    # l'attribut statique punlic 'liste' peut être accédé
                        # en préfixant avec un objet de la classe Point.
```

```
3
x: 1.000000, y: 1.000000
x: 2.000000, y: 2.000000
3
```

Méthodes statiques

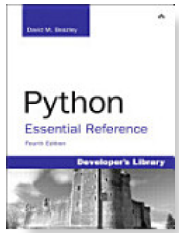
```
class Point:
    __liste = []    # définition de l'attribut de classe (statique) public 'liste'

    def __init__(self, x=0, y=0):
        self.__x = x  # attribut privé
        self.__y = y  # attribut privé
        Point.__liste.append(self) # utilisation d'un attribut statiques : préfixer
                                   # son nom par le nom de la classe.

    def __str__(self):        # redéfinition (surcharge) de la fonction str()
        return "x: {0:f}, y: {1:f}".format(self.__x, self.__y)

    @staticmethod             # le décorateur @staticmethod permet de définir
    def nbPoint():            # les méthodes statiques => Pas d'argument self !
        return len(Point.__liste)

Point(1,1)                    # création de 2 objets Point sans nom
Point(2,2)
p1=Point(3,3)                 # création d'un objet Point nommé p1
# print(len(Point.__liste))  -> ERREUR : attribut statique 'nbPt' est privé !
print(Point.nbPoint())        # appel de la méthode statique
```

```
3
```

Références bibliographiques

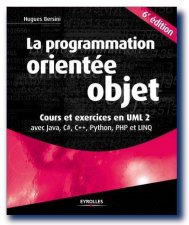
https://docs.python.org/index.html
https://openclassrooms.com/courses/apprenez-a-programmer-en-python

Apprenez à programmer en Python
Vincent Le Goff
Simple IT éd. (Le livre du zéro)
ISBN 979-10-90085-03-9
≈25 €



Python Essential Reference
David M. Beazley
Addison Wesley
ISBN 0-672-32978-4

Apprendre à programmer avec Python 3
Gérard Swinnen
Télécharger le PDF



L'orienté objet, Cours et exercices UML 2, Python, C++...
H. Bersini
EYROLLES
ISBN 2212120842

✉ jean-luc.charles@ensam.eu

✉ eric.ducasse@ensam.eu