POO

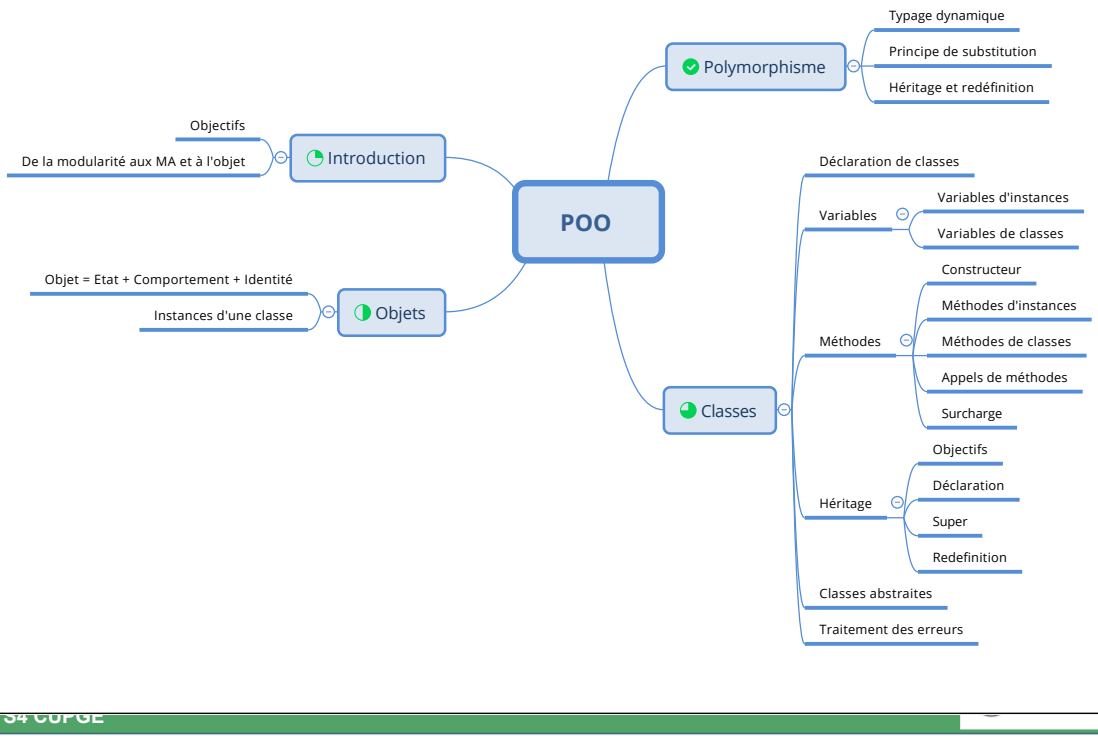
Programmation Orientée Objet

UNIVERSITÉ TOULOUSE III PAUL SABATIER

1

1

POO : Introduction



L2 S4 CUPGE

2

2

Objectif

- **Introduction à la Programmation Orientée Objet**
  - **Prérequis : programmation impérative**
    - Structure de contrôles, Sous-programmes, TAD
  - **Qualité du logiciel**
    - Réutilisabilité, fiabilité, robustesse
    - Qualité : difficile à mettre en oeuvre
      - 30% du coût en développement
      - 70% en maintenance !!
    - Coût de la non qualité
      - Système de réservation de United Airlines
      - Logiciels de gestion (livraison de journaux)
      - Logiciels temps réels embarqués (F16)
      - Logiciels temps réels sols (lune)

Modularité

- **Autour des données ou autour des traitements ?**
  - **Programme = Actions sur des Données**
    - Décomposition classique par les actions (fonctionnelle, sous-prog)
  - **Ou**
    - Décomposition par les données (modules, classes)

Décomposition fonctionnelle descendante
si langage fonctionnel

Décomposition par les données
si langage à objets

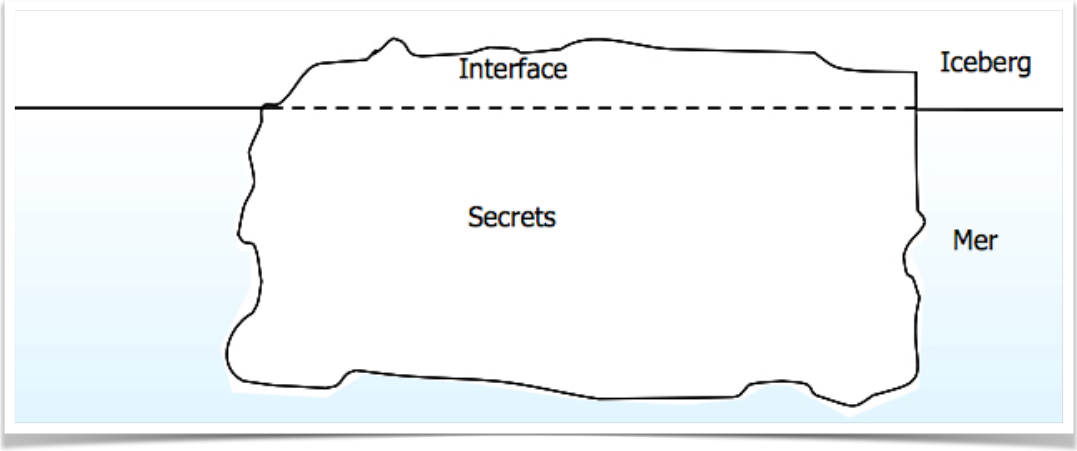
Oui à la composition ascendante

- **Partir des objets du système**
  - **Créer un module pour chaque type d'objets qui gère les données et les services**
  - **Développer des services compatibles**
  - **Penser réutilisabilité !**

Ne pas définir a priori ce que fait le système : définir SUR QUOI il agit !

Mise en oeuvre de la modularité

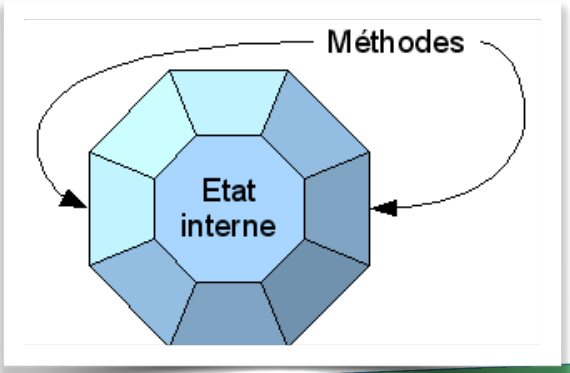
- **Interface publique**
- **Le reste doit (devrait, pb en python) rester caché !**



Objets : définition

Objet = Etat + Comportement + Identité

- **Etat interne**
- **Comportement = compétences**
  - **Agit sur l'état interne**



Objet : Instance de classe

- **Identité UNIQUE**
  - **Référence de la zone mémoire allouée lors de la création de l'objet**
- **Etat = valeurs des attributs**

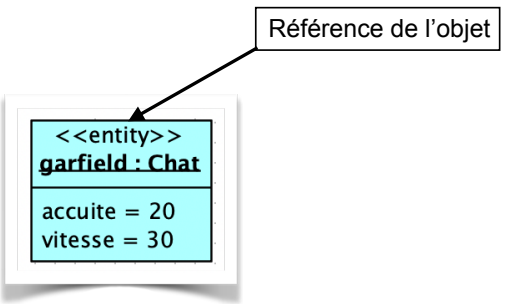
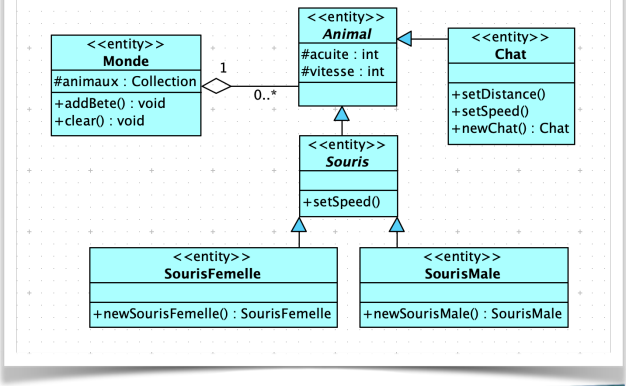


Diagramme de classe (à venir)



Classes : Déclaration

- **Classe simple**

```
class MyClassName :
        # attributs
        # constructeurs
        # méthodes
```

- **Héritage simple**

```
class MyClassName(ClasseParente) :
        ...
```

- **Héritage multiple**

```
class MyClassName(Parent1, Parent2, ...) :
        ...
```


Constructeur

- **Création d'une instance**

constructeur = allocation + initialisation

- **Java :**

```
public class Velo {
  public Velo ( ) { … }
  public Velo (String type) { …  }
}
// client
v1 = new Velo("VTT);
```

- **Python :**

```
class Pile: # sans paramètre
    # constructeur de pile
    def __init__(self):
        …
class Velo: # avec paramètre(s)
    # constructeur de vélo
    def __init__(self, type):
        …
```

```
# client
p1 = Pile()
v1 =
Velo("VTT")
```

Variable d'instance

- **Attribut dont la valeur est stockée dans chaque instance**
  - **Règle : créer les attributs dans le constructeur**
  - **Java : différents niveaux de visibilité**
    - public, protected ou private (défaut package à éviter)
  - ⚠️ **Python :** public ou protected == simple convention

```
class Pile:
    # créer une pile vide
    def __init__(self):
        # attribut public
        self.items = []
```

```
class Velo:
    # créer un vélo
    def __init__(self, type):
        # attribut "protected"
         # mais accessible
        self._type = type
```

```
# chez le client
p1 = Pile()
v1 = Velo("VTT")
print("type du vélo : "+ v1._type)
```


Variable d'instance

- **Attribut dont la valeur est stockée dans chaque instance**
  - **Python :**
    - attributs private (name mangling)
      - meilleure protection mais contournable

```
class Velo:
    # créer un vélo
    def __init__(self, type):
        self.__type = type
```

```
# chez le client
v1 = Velo("VTT")
print("type du vélo : "+ v1.__type) # erreur
print("type du vélo : "+ v1._Velo__type) # ok !!!
v1._Velo__type = 12 #!!!!!!!!!!!!!!!!!!!!!!!
```

Classe Etudiant

```
class Etudiant:
    def __init__(self, nom: str, dateNais: Date):
        # attributs "protected"
        self._nom = nom
        # self.date = date !!!!!!!! Erreur à ne pas faire !!!!!!!!!
        self._dateNaissance = dateNais.copieCons()
        self._niveauLFlex = 0
    def incrNiveauLFlex(self):
        self._niveauLFlex += 1
    def __eq__(self, p):
        return self._dateNaissance == p._dateNaissance
    def __lt__(self, p):
        if (self == p):
            return 0
        if (self._dateNaissance.__lt__(p._dateNaissance)):
            return +1 # plus vieux donc plus grand
        return -1
    def __str__(self):
        return "[{}, {}, {}]".format(self._nom, self._niveauLFlex,
            self._dateNaissance)
```


Variable "de classe"

- Attribut accessible à toutes les instances de la classe
  - ⚠️ Pour stocker des valeurs qui -devraient- rester constantes
  - Java : "vraies" variables de classes (et méthodes de classes)
  - Python : pas de partage entre instances (cf EtudiantSHN)

```
class EtudiantSHN:
    # variable de classe
    __NOM = "SHN"
    def __init__(self, nom: str, sport: str, date: Date):
        # …
    # …
# client
  d = Date(31, 12, 2000)
  julie = EtudiantSHN("Julie", "athletisme", d)
  print(julie)
  d = d.lendemain()
  jules = EtudiantSHN("Jules", "TKD", d)
  print(jules)
  jules._EtudiantSHN__NOM = "basket"
  print(julie)
  print(jules)
```

Méthodes d'instances

- Java : surcharge des méthodes et des constructeurs
  - Même nom de méthode, paramètres différents
- Python : pas de méthodes surchargées
- Solution : paramètres facultatifs

```
class Velo:
  # créer un vélo avec son type et son propriétaire (facultatif)
  def __init__(self, type: str, proprietaire: Etudiant = None):
    self.__type = type
    self.__proprietaire = proprietaire

  def getProprietaire(self):
    return self.__proprietaire
  def setProprietaire(self, proprio: Etudiant):
    self.__proprietaire = proprio

  def __str__(self):
    return "[type : {}, propriétaire : {}]".format(self.__type,
        self.__proprietaire)
```


Appel des méthodes d'instance

- Méthodes appliquées à l'instance (invoquée)
- Exemple de code client
  - Résultat de l'exécution ?

```
d = Date(1, 1, 1990)
julie = Etudiant("Julie", d)
d = Date(31, 12, 2000)
bob = Etudiant("Bob", d)
print(julie)
print(bob)
vJulie = Velo("VTT", julie)
vBob = Velo("Gravel", bob)
v = Velo("Route")
print(vJulie)
print(vBob)
print(v)
v.setProprietaire(bob)
print(v)
julie.setNiveauLFlex(1)  # Julie modifie son âge
print("Après incrément : " + str(julie))
```

Constructeur par copie

- **Pour créer un duplicata de l'instance (ici p1)**
  - ⚠️ **Pas de copies de références !!!**
  - **En Java**
    - surcharge du constructeur avec une instance en entrée
    - retourne un duplicata de cette instance
      - Point p2 = new Point(p1);
  - **En Python :**
    - méthode spécifique
      - p2 = p1.copieCons()
- **Créer des classes en ajoutant les méthodes : (a minima)**
  - **copieCons**
  - **__str__**
  - **__eq__**

Etudiant

```
class Etudiant:
    def __init__(self, nom: str, dateNais: Date):
        self._nom = nom
        # attention, pas de copies de références !!
        self._dateNaissance = dateNais.copieCons()
        self._niveauLFlex = 0
    def copieCons(self):
        return Etudiant(self._nom, self._dateNaissance)

    def setNiveauLFlex(self, niveau = 0):
        self._niveauLFlex = niveau
    def __eq__(self, p):
        return self._dateNaissance == p._dateNaissance
    def __lt__(self, p):
        if (self == p):
            return 0
        if (self._dateNaissance.__lt__(
                p._dateNaissance)):  # plus vieux donc plus grand
            return +1
        return -1

    def __str__(self):
        return "[nom : {}, niveau : {}, date de naissance :
{}]".format(self._nom, self._niveauLFlex, self._dateNaissance)
```

Gestion des erreurs

- **Vues en NSI**
  - **Assert**
    - Programmation par contrat
    - Erreurs pouvant être évitées, corrigées

```
def depiler(self):
    assert (not self.estVide()), "Pile vide, impossible de dépiler"
    …
```

  - **Exceptions**
    - Erreurs ne pouvant être évitées
    - Entrées/sorties, IHM
  - **Tests**
    - unittest

Clauses else et finally

- **Clause else pour gérer le cas nominal**
- **Clause finally toujours exécutée**

```
def saisieProtegee(borneInf, borneSup):
    ok = False
    cpt = 0
    while not ok:
        try:
            cpt += 1
            val = int(input(str(borneInf)+"<= v <="+str(borneSup)))
            if (val < borneInf) or (borneSup < val):
                raise ValueError
        except ValueError:
            print("valeur erronée, recommencer")
        except Exception:
            print("Entrer un entier !!")
        else:
            ok = True
        finally:
            print("compteur d'essais : "+str(cpt))
    return val
```

Héritage

- **Hériter :**
  - **Classe dérivée (fille, héritière) <=> disposer des attributs et méthodes de la classe parente**

- **Java : héritage simple**
  - **Une classe hérite *d'une seule* classe**
  - **Héritage implicite de *Object***
  - **Interfaces (*implements)* permet de décrire des comportements**

- **Python : héritage multiple**
  - **Héritage implicite de *object* (fonctions de la forme __xx__)**
  - ⚠ **Conflits de noms possibles entre attributs des parents !!!**

Héritage

- **Déclaration**

```
class MyClassName (ClasseParente1[, ClasseParente2, ...]):
        ... # code
```

- **Constructeur**
  - **Appel OBLIGATOIRE du constructeur du parent en première ligne**
    - super().__init__(nom, date)
    - Etudiant.__init__(self, nom, date)

```
class EtudiantSHN(Etudiant) :
    __NOM = "SHN"  # variable de classe
    def __init__(self, nom: str, sport: str, date: Date):
        # !!!! toujours la première instruction
        #du constructeur de la classe dérivée !!!!
        super().__init__(nom, date)
        self.__sport = sport    # variable d'instance
```

Redéfinition

- **Si B hérite de A alors**
  - **B hérite de tous les attributs et méthodes du parent**
  - **B hérite de toutes les méthodes *pré*définies de *objet***
    - __str__ : retourne par défaut l'@ hexa de l'instance en mémoire
    - __eq__ : compare les @ en mémoire du **self** et du paramètre
- ***Redéfinir* toutes les méthodes qui ne conviennent pas !!**
- **——>>>>> ATTENTION : surcharge != redéfinition**
  - **surcharge : même nom mais signature *différente***
  - **redéfinition : *même* signature**

Redéfinition

```
class A:
    def __init__(self, val):
        self.__valA = val
    def getVal(self):
        return self.__valA
    def __str__(self): # redéfinition de la méthode de objet
        return str(self.__valA)

class B(A):
    def __init__(self, val1, val2):
        super().__init__(val1)
        self.__valB = val2
    def getVal(self): # redéfinition de la méthode de A
        return (super().getVal(), self.__valB)
    def __str__(self): # redéfinition de la méthode de A
        return "({}, {})".format(super(B, self).__str__(),
                                 str(self.__valB))
# test
if __name__ == '__main__':
    a = A(12)
    print(a)
    b = B(12, 31)
    print(b)
```

Héritage multiple

- Constructeur doit appeler les constructeurs des parents
- Conflits de noms possibles

```
class Sport:
    def __init__(self, nom: str):
        self.__sport = nom

    def __str__(self):
        return self.__sport

class EtudiantSHN_V2(Etudiant, Sport) :
    # variable de classe
    NOM = "SHN"
    def __init__(self, nom: str, sport: str, date: Date):
        # toujours les 1ère instructions du cons de la classe dérivée
        Etudiant.__init__(self, nom, date)
        Sport.__init__(self, sport)

    def __str__(self): # __str__
        return "[{}, {}, sport : {}]".format(self.NOM,
          Etudiant.__str__(self), Sport.__str__(self))
```


Méthodes et classes abstraites

- Hérite de ABC
- Si *au moins 1* méthode abstraite alors classe abstraite
- Une classe abstraite *ne peut pas* être instanciée
- Les classes concrètes *doivent* mettre en œuvre les méthodes abstraites

```
from abc import ABC, abstractmethod
# hérite de Abstract Base Class
class Crispy(ABC):
    @abstractmethod
    def sound(self):
        return
```

```
# classes concrètes
class Snap(Crispy):
    def sound(self):
        print("snap", end=' ')
class Pop(Crispy):
    def sound(self):
        print("pop", end=' ')
class Crakle(Crispy):
    def sound(self):
        print("crack", end=' ')
```

Modélisation de Bibliothèque