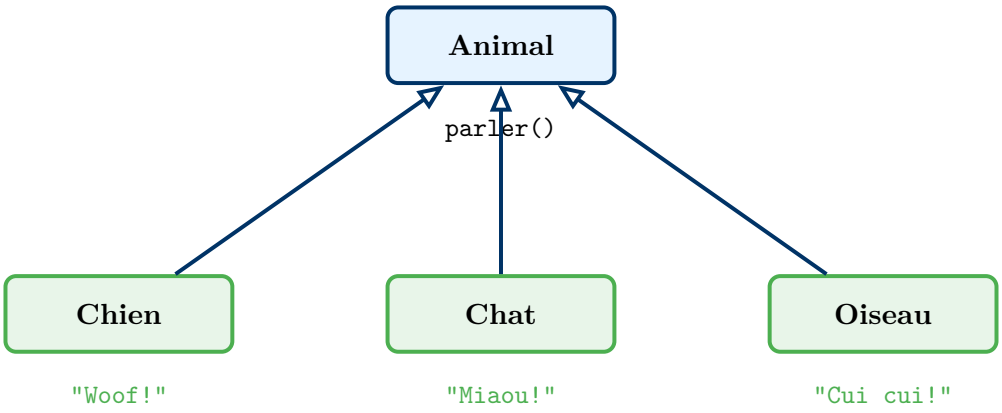
M103 - Programmation Orientée Objet

Séance 4 : Le Polymorphisme

ISTA Mirleft - Développement Digital (1ère Année) - 2025/2026



Une méthode, plusieurs comportements

Table des matières

1 Introduction au Polymorphisme

1.1 Rappel : L'Héritage

Dans la séance précédente, nous avons appris que l'**héritage** permet à une classe enfant d'hériter des attributs et méthodes de sa classe parent.

```python
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def parler(self):
        print("L'animal fait un son")

class Chien(Animal):
    pass  # Herite tout de Animal

rex = Chien("Rex")
rex.parler()  # L'animal fait un son
```

Listing 1 – Rappel - Héritage simple

Question : Mais que faire si nous voulons que le chien aboie au lieu de faire "un son" ?

1.2 Définition du Polymorphisme

Définition : Le Polymorphisme

Le **polymorphisme** (du grec "plusieurs formes") est la capacité d'une méthode à se comporter **différemment** selon la classe qui l'implémente.

Principe : Une même méthode (même nom) peut avoir des comportements différents dans les classes filles.

Avantages :

- Code plus flexible et extensible
- Traitement uniforme d'objets différents
- Facilite l'ajout de nouvelles classes
- Principe fondamental de la POO

1.3 Types de Polymorphisme

Type	Description
Polymorphisme par héritage	Redéfinition d'une méthode dans une classe fille (Override)
Polymorphisme par surcharge	Même méthode avec différents paramètres (limité en Python)
Duck Typing	Si ça marche comme un canard... (spécifique à Python)

2 Redéfinition de Méthodes (Override)

2.1 Principe de la Redéfinition

La **redéfinition** (ou *override*) consiste à réécrire une méthode héritée dans la classe enfant pour modifier son comportement.

```python
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def parler(self):
        print(f"{self.nom} fait un son")

class Chien(Animal):
    def parler(self):  # Redefinition !
        print(f"{self.nom} aboie : Woof woof!")

class Chat(Animal):
    def parler(self):  # Redefinition !
        print(f"{self.nom} miaule : Miaou!")

class Oiseau(Animal):
    def parler(self):  # Redefinition !
        print(f"{self.nom} chante : Cui cui!")

# Utilisation
rex = Chien("Rex")
minou = Chat("Minou")
piou = Oiseau("Piou")

rex.parler()    # Rex aboie : Woof woof!
minou.parler()  # Minou miaule : Miaou!
piou.parler()   # Piou chante : Cui cui!
```

Listing 2 – Redéfinition de méthode

Points Clés de la Redéfinition

- La méthode dans la classe fille a le **même nom** que dans la classe parent
- La méthode de la classe fille **remplace** celle du parent
- Chaque classe fille peut avoir sa propre implémentation
- L'appel de la méthode dépend du **type réel** de l'objet

2.2 Polymorphisme en Action

Le vrai pouvoir du polymorphisme se révèle quand on traite des objets différents de manière uniforme :

```python
# Creer une liste d'animaux differents
animaux = [
    Chien("Rex"),
    Chat("Minou"),
    Oiseau("Piou"),
    Chien("Max"),
    Chat("Felix")
]

# Traitement uniforme - meme code pour tous !
```

```python
print("=== Tous les animaux parlent ===")
for animal in animaux:
    animal.parler()  # Chaque animal parle a sa maniere

# Sortie :
# Rex aboie : Woof woof!
# Minou miaule : Miaou!
# Piou chante : Cui cui!
# Max aboie : Woof woof!
# Felix miaule : Miaou!
```

Listing 3 – Polymorphisme avec une liste d'objets

Pourquoi c'est puissant ?

Sans polymorphisme, il faudrait écrire :

```python
for animal in animaux:
    if isinstance(animal, Chien):
        print("Woof!")
    elif isinstance(animal, Chat):
        print("Miaou!")
    elif isinstance(animal, Oiseau):
        print("Cui cui!")
```

Avec le polymorphisme, une seule ligne suffit : `animal.parler()`

3 Étendre une Méthode avec super()

3.1 Réutiliser le Code du Parent

Parfois, on ne veut pas **remplacer** complètement la méthode parent, mais l'**étendre** en ajoutant du comportement.

```python
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def se_presenter(self):
        print(f"Bonjour, je suis {self.nom}, j'ai {self.age} ans.")

class Etudiant(Personne):
    def __init__(self, nom, age, filiere):
        super().__init__(nom, age)  # Appel constructeur parent
        self.filiere = filiere

    def se_presenter(self):
        super().se_presenter()  # Appel methode parent
        print(f"J'etudie en {self.filiere}.")

class Formateur(Personne):
    def __init__(self, nom, age, specialite):
        super().__init__(nom, age)
        self.specialite = specialite

    def se_presenter(self):
        super().se_presenter()  # Appel methode parent
        print(f"J'enseigne {self.specialite}.")

# Utilisation
ahmed = Etudiant("Ahmed", 20, "Developpement Digital")
sara = Formateur("Sara", 35, "Python et POO")

ahmed.se_presenter()
# Bonjour, je suis Ahmed, j'ai 20 ans.
# J'etudie en Developpement Digital.

print()

sara.se_presenter()
# Bonjour, je suis Sara, j'ai 35 ans.
# J'enseigne Python et POO.
```

Listing 4 – Extension de méthode avec super()

Quand utiliser super() dans une méthode ?

- **Utilisez super()** quand vous voulez **ajouter** du comportement à la méthode parent
- **N'utilisez pas super()** quand vous voulez **remplacer complètement** le comportement
- **super()** peut être appelé au **début**, au **milieu** ou à la **fin** de la méthode enfant

3.2 Position de super() dans la Méthode

```python
class Parent:
```

```
    def action(self):
        print("Action du parent")

class Enfant1(Parent):
    def action(self):
        super().action()  # Parent d'abord
        print("Action de l'enfant")

class Enfant2(Parent):
    def action(self):
        print("Action de l'enfant")
        super().action()  # Parent ensuite

class Enfant3(Parent):
    def action(self):
        print("Debut enfant")
        super().action()  # Parent au milieu
        print("Fin enfant")

# Test
Enfant1().action()
# Action du parent
# Action de l'enfant

Enfant3().action()
# Debut enfant
# Action du parent
# Fin enfant
```

Listing 5 – Différentes positions de super()

4 Surcharge d'Opérateurs

4.1 Qu'est-ce que la Surcharge d'Opérateurs ?

Python permet de définir le comportement des opérateurs (+, -, ==, <, etc.) pour nos propres classes grâce aux **méthodes spéciales** (aussi appelées *méthodes magiques* ou *dunder methods*).

Méthodes Spéciales pour les Opérateurs

Opérateur	Méthode	Description
==	`__eq__(self, other)`	Égalité
!=	`__ne__(self, other)`	Différence
<	`__lt__(self, other)`	Inférieur à
<=	`__le__(self, other)`	Inférieur ou égal
>	`__gt__(self, other)`	Supérieur à
>=	`__ge__(self, other)`	Supérieur ou égal
+	`__add__(self, other)`	Addition
-	`__sub__(self, other)`	Soustraction
*	`__mul__(self, other)`	Multiplication
len()	`__len__(self)`	Longueur

4.2 Exemple : Comparer des Objets avec `__eq__()`

```python
class Produit:
    def __init__(self, nom, prix, reference):
        self.nom = nom
        self.prix = prix
        self.reference = reference

    def __eq__(self, other):
        """Deux produits sont egaux s'ils ont la meme reference"""
        if isinstance(other, Produit):
            return self.reference == other.reference
        return False

    def __str__(self):
        return f"{self.nom} ({self.reference}) - {self.prix} DH"

# Utilisation
p1 = Produit("iPhone 15", 12000, "APL-IP15")
p2 = Produit("iPhone 15 Pro", 15000, "APL-IP15")  # Meme reference
p3 = Produit("Samsung S24", 10000, "SAM-S24")

print(p1 == p2)  # True (meme reference)
print(p1 == p3)  # False (references differentes)

# Sans __eq__, Python comparerait les adresses memoire
# et p1 == p2 serait False !
```

Listing 6 – Surcharge de l'opérateur ==

4.3 Exemple : Addition d'Objets avec `__add__()`

```python
class Vecteur:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __add__(self, other):
        """Addition de deux vecteurs"""
        if isinstance(other, Vecteur):
            return Vecteur(self.x + other.x, self.y + other.y)
        raise TypeError("Impossible d'additionner avec ce type")

    def __str__(self):
        return f"Vecteur({self.x}, {self.y})"

# Utilisation
v1 = Vecteur(3, 4)
v2 = Vecteur(1, 2)
v3 = v1 + v2  # Appelle v1.__add__(v2)

print(v3)  # Vecteur(4, 6)
```

Listing 7 – Surcharge de l'opérateur +

4.4 Exemple : Comparaison avec ___lt___() et ___gt___()

```
class Etudiant:
    def __init__(self, nom, moyenne):
        self.nom = nom
        self.moyenne = moyenne

    def __lt__(self, other):
        """Compare par moyenne (pour le tri)"""
        return self.moyenne < other.moyenne

    def __gt__(self, other):
        return self.moyenne > other.moyenne

    def __str__(self):
        return f"{self.nom}: {self.moyenne}/20"

# Utilisation
etudiants = [
    Etudiant("Ahmed", 15.5),
    Etudiant("Sara", 17.0),
    Etudiant("Karim", 14.0)
]

# Trier automatiquement grace a __lt__
etudiants_tries = sorted(etudiants)
for e in etudiants_tries:
    print(e)
# Karim: 14.0/20
# Ahmed: 15.5/20
# Sara: 17.0/20

# Comparaisons directes
print(etudiants[0] < etudiants[1])  # True (15.5 < 17.0)
```

Listing 8 – Surcharge des opérateurs de comparaison

5 Duck Typing (Typage Canard)

5.1 Principe du Duck Typing

Duck Typing

"Si ça marche comme un canard, si ça fait coin-coin comme un canard, alors c'est un canard."

En Python, on ne vérifie pas le **type** d'un objet, mais s'il possède les **méthodes/attributs** nécessaires.

```python
class Canard:
    def parler(self):
        print("Coin coin!")

    def marcher(self):
        print("Le canard marche")

class Personne:
    def parler(self):
        print("Bonjour!")

    def marcher(self):
        print("La personne marche")

class Robot:
    def parler(self):
        print("Bip bip!")

    def marcher(self):
        print("Le robot marche")

def faire_parler_et_marcher(chose):
    """Cette fonction accepte N'IMPORTE QUEL objet
    qui a les methodes parler() et marcher()"""
    chose.parler()
    chose.marcher()

# Tous ces appels fonctionnent !
faire_parler_et_marcher(Canard())
faire_parler_et_marcher(Personne())
faire_parler_et_marcher(Robot())

# Meme si ces classes n'ont AUCUN lien d'heritage !
```

Listing 9 – Duck Typing en action

Avantages du Duck Typing

- Plus de flexibilité : pas besoin d'héritage commun
- Code plus simple : pas de vérification de type
- Facilite les tests : on peut créer des "mocks" facilement

Inconvénient

Si un objet n'a pas la méthode requise, l'erreur survient à l'**exécution** (pas à la compilation). Il faut donc bien tester son code !

6 Exemple Pratique Complet

6.1 Système de Calcul de Salaires

```python
class Employe:
    """Classe de base pour tous les employes"""

    def __init__(self, nom, salaire_base):
        self.nom = nom
        self.salaire_base = salaire_base

    def calculer_salaire(self):
        """A redefinir dans les classes filles"""
        return self.salaire_base

    def __str__(self):
        return f"{self.nom} - Salaire: {self.calculer_salaire()} DH"

class Developpeur(Employe):
    """Les developpeurs ont une prime de projet"""

    def __init__(self, nom, salaire_base, prime_projet=0):
        super().__init__(nom, salaire_base)
        self.prime_projet = prime_projet

    def calculer_salaire(self):
        return self.salaire_base + self.prime_projet

class Commercial(Employe):
    """Les commerciaux ont une commission sur ventes"""

    def __init__(self, nom, salaire_base, ventes=0, taux_commission=0.05):
        super().__init__(nom, salaire_base)
        self.ventes = ventes
        self.taux_commission = taux_commission

    def calculer_salaire(self):
        commission = self.ventes * self.taux_commission
        return self.salaire_base + commission

class Manager(Employe):
    """Les managers ont un bonus base sur la taille de l'equipe"""

    def __init__(self, nom, salaire_base, taille_equipe=0):
        super().__init__(nom, salaire_base)
        self.taille_equipe = taille_equipe

    def calculer_salaire(self):
        bonus = self.taille_equipe * 500  # 500 DH par membre
        return self.salaire_base + bonus

# === UTILISATION ===
employes = [
    Developpeur("Ahmed", 12000, prime_projet=3000),
    Commercial("Sara", 8000, ventes=100000),
    Manager("Karim", 15000, taille_equipe=5),
    Developpeur("Fatima", 11000, prime_projet=2000),
    Commercial("Omar", 8000, ventes=50000)
]

```

```python
# Calcul de la masse salariale (polymorphisme!)
print("=== Fiche de Paie ===")
total = 0
for emp in employes:
    print(emp)  # Appelle __str__ qui appelle calculer_salaire()
    total += emp.calculer_salaire()

print(f"\nMasse salariale totale: {total} DH")
```

Listing 10 – Système de paie polymorphe

Sortie :

```
=== Fiche de Paie ===
Ahmed - Salaire: 15000 DH
Sara - Salaire: 13000 DH
Karim - Salaire: 17500 DH
Fatima - Salaire: 13000 DH
Omar - Salaire: 10500 DH

Masse salariale totale: 69000 DH
```

7 Bonnes Pratiques

Bonne Pratique	Explication
Respecter le contrat	La méthode redéfinie doit avoir un comportement cohérent avec la méthode parent
Utiliser super() judicieusement	Appeler super() quand on veut étendre, pas quand on veut remplacer
Documenter les redéfinitions	Expliquer pourquoi la méthode est redéfinie
Tester le polymorphisme	Vérifier que toutes les classes filles fonctionnent correctement
Éviter les conditions sur le type	Préférer le polymorphisme aux `if isinstance(...)`

8 Erreurs Fréquentes

8.1 Erreur 1 : Oublier de maintenir la compatibilité

```python
# INCORRECT
class Animal:
    def manger(self, nourriture):
        print(f"Mange {nourriture}")

class Chien(Animal):
    def manger(self):  # Signature differente !
        print("Mange des croquettes")

# Probleme: le code qui attend manger(nourriture) va echouer

# CORRECT
class Chien(Animal):
    def manger(self, nourriture="croquettes"):
        print(f"Le chien mange {nourriture}")
```

Listing 11 – Erreur - Signature incompatible

8.2 Erreur 2 : Ne pas vérifier le type dans `__eq__()`

```python
# INCORRECT
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y  # Crash si other n'a
            pas x/y

# CORRECT
class Point:
    def __eq__(self, other):
        if not isinstance(other, Point):
```

```
14                 return False
15             return self.x == other.x and self.y == other.y
```

Listing 12 – Erreur - Comparaison sans vérification

9 Résumé

Ce qu'il faut retenir

1. **Polymorphisme :**
   - Une même méthode peut avoir des comportements différents
   - Permet de traiter des objets différents de manière uniforme
2. **Redéfinition (Override) :**
   - Réécrire une méthode héritée dans la classe fille
   - La méthode de la classe fille remplace celle du parent
3. **Extension avec super() :**
   - `super().methode()` appelle la méthode du parent
   - Permet d'ajouter du comportement sans tout réécrire
4. **Surcharge d'opérateurs :**
   - Définir le comportement des opérateurs (+, ==, <, etc.)
   - Utiliser les méthodes spéciales (___eq___, ___add___, etc.)
5. **Duck Typing :**
   - Python ne vérifie pas le type, mais les méthodes disponibles
   - Plus de flexibilité, mais attention aux erreurs d'exécution