POO en Python

Héritage & Polymorphisme

Partie 2/3

Anis SAIED

IPEIN

2025/26

Héritage entre classes

L'**héritage**

- Est une caractéristique fondamentale de la programmation orientée objet (POO).
- Permet de créer de nouvelles classes basées sur des classes existantes.
- Favorise la réutilisation du code et la structuration des classes en hiérarchies.
- En Python, l'héritage s'effectue en spécifiant la **classe parente** entre parenthèses lors de la définition de la **classe enfant**.

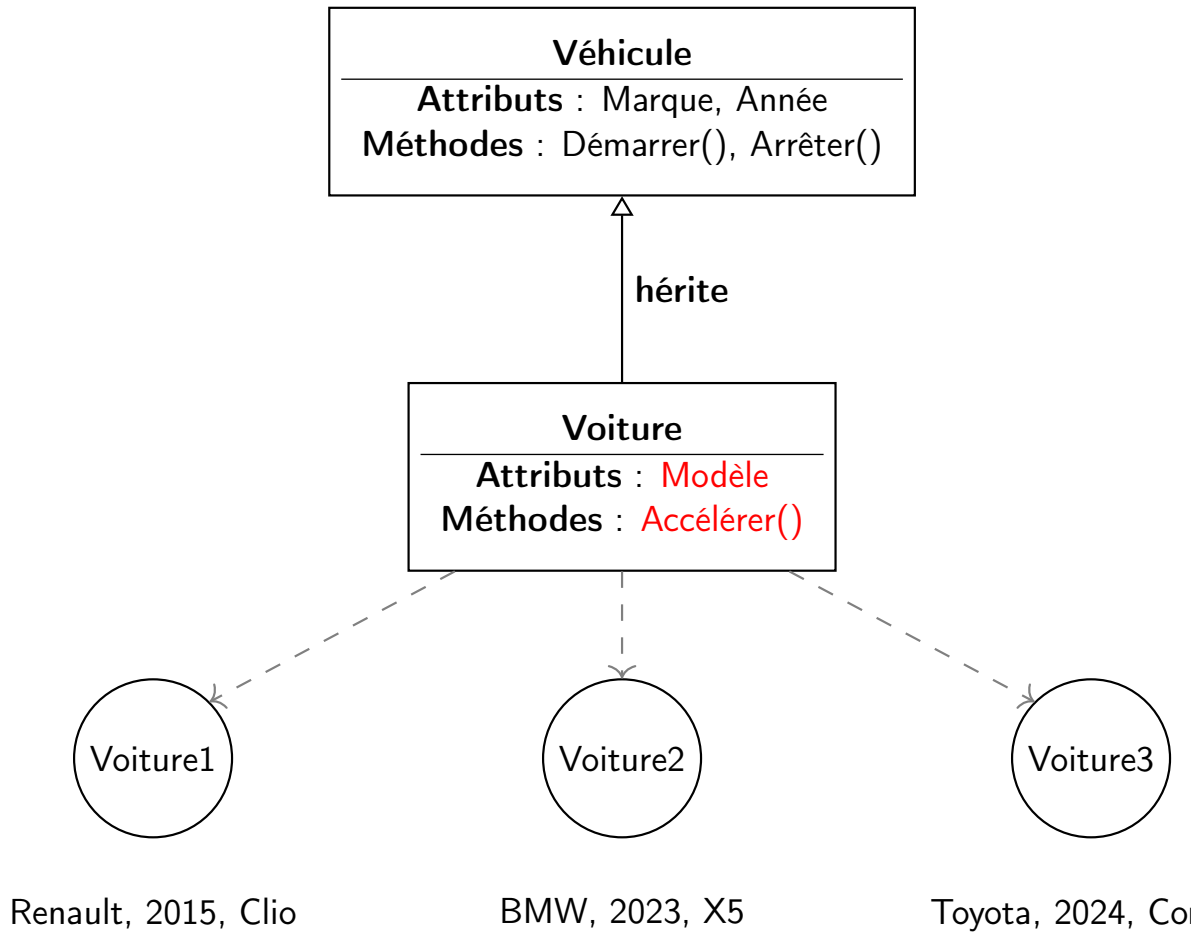
```
1 class Parent:
2     # Attributs et méthodes de la classe parente
3
4 class Enfant(Parent):
5     # Attributs et méthodes spécifiques à la classe enfant
```

Héritage entre Classes

Exemple 1

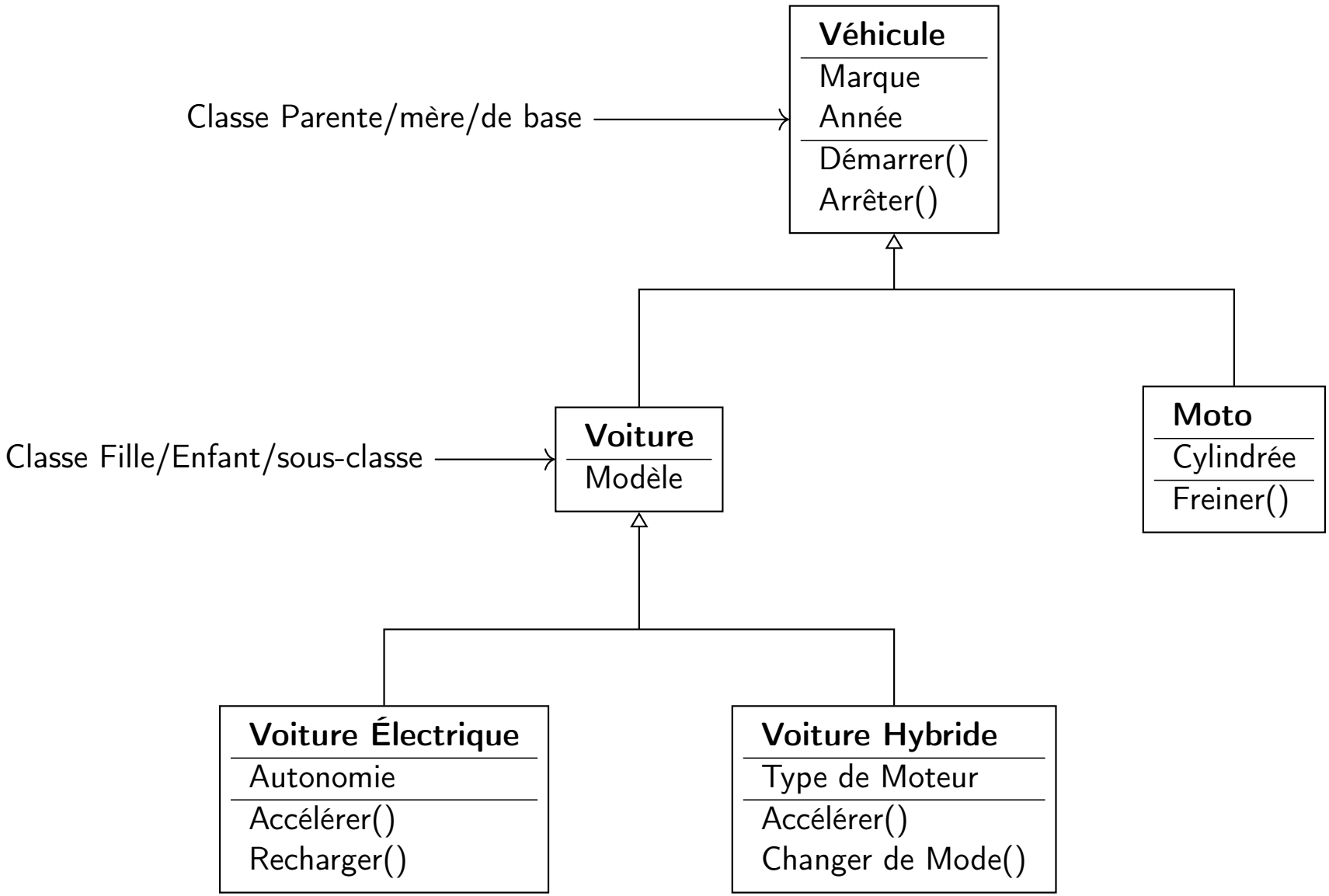
Considérons une **classe parente** appelée `Véhicule` avec les attributs `marque` et `année`.

La **classe enfant** appelée `Voiture` **hérite** la classe `Véhicule` et ajoute un attribut `modèle`.



Héritage entre Classes

Exemple 2: Diagramme de classes



Héritage entre classes

Implémentation

```python
class Véhicule: # classe de base
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee

class Voiture(Véhicule): # classe dérivée
    def __init__(self, marque, annee, modele):
        # Reutilisation de code
        super().__init__(marque, année) # self est passé implicitement
        # Ajouter des attributs spécifiques
        self.modele = modele

# Créez une instance de Voiture et affichez ses attributs
voiture1 = Voiture("Toyota", 2022, "Camry")
print("Marque:", voiture1.marque)
print("Année:", voiture1.annee)
print("Modèle:", voiture1.modele)
```

super() :

- fait référence à la classe parent (ou mère) la plus proche.
- permet d'appeler une méthode de la classe parente depuis une classe enfant pour étendre ou modifier son comportement.

Exercice 1

1. Définissez une classe **Forme** avec une méthode **aire** qui renvoie 0.
2. Définissez la classe **Point**, qui est une sous-classe de Forme. Ajoutez des attributs **x** et **y** pour représenter les coordonnées d'un point et une méthode **afficher** qui affiche les coordonnées.
3. Créez une classe **Cercle**, qui est une sous-classe de Point. Ajoutez un attribut **rayon** pour représenter le rayon du cercle et une méthode **aire** pour calculer l'aire du cercle.
4. Créez une classe **Rectangle**, qui est également une sous-classe de Point. Ajoutez des attributs **largeur** et **hauteur** pour représenter les dimensions du rectangle et une méthode **aire** pour calculer l'aire du rectangle.
5. Créez un objet de chaque classe (Point, Cercle et Rectangle) et utilisez les méthodes `aire` et `afficher` pour afficher leurs aires respectives et leurs coordonnées.

Exercice 1 I

Solution

```
1  class Forme:
2      def aire(self):
3          return 0 # valeur par défaut
4
5  class Point(Forme):
6      def __init__(self, x, y):
7          self.x = x
8          self.y = y
9
10     def afficher(self):
11         print("Point : ({}, {})".format(self.x, self.y))
12
13 import math
14
15 class Cercle(Point):
16     def __init__(self, x, y, rayon):
17         super().__init__(x, y)
18         self.rayon = rayon
19
20     def aire(self):
21         return math.pi * self.rayon ** 2
```

Exercice 1 II

Solution

```
22 class Rectangle(Point):
23     def __init__(self, x, y, largeur, hauteur):
24         super().__init__(x, y)
25         self.largeur = largeur
26         self.hauteur = hauteur
27     def aire(self):
28         return self.largeur * self.hauteur
29 
30 point = Point(2, 3)
31 cercle = Cercle(0, 0, 5)
32 rectangle = Rectangle(1, 1, 4, 3)
33 
34 point.afficher()
35 print("Aire du point : {:.2f}".format(point.aire()))
36 
37 cercle.afficher()
38 print("Aire du cercle : {:.2f}".format(cercle.aire()))
39 
40 rectangle.afficher()
41 print("Aire du rectangle : {:.2f}".format(rectangle.aire()))
```

Relations "Est-un" & "A-un"

Définition

En programmation orientée objet (POO), deux concepts clés pour modéliser les relations entre objets sont l'**héritage** et la **composition** :

- L'héritage permet de créer des classes dérivées (enfants) à partir de classes de base (parents), établissant ainsi une relation "**est un**" où la classe dérivée est un type spécialisé de la classe de base.
  - Exemple : Une "Voiture" **est un** "Véhicule".

- La relation "**a un**" est implémentée via la composition, qui consiste à inclure des objets d'une classe dans une autre, où la classe contenant l'objet est responsable de sa création et de sa gestion.
  - Exemple : Une "Voiture" **a un** "Moteur".

Relations "Est-un" & "A-un"

Syntaxe

Héritage (Relation "Est-un") :

```python
class ClasseParent:
    # Attributs et méthodes de la classe parente
class ClasseEnfant(ClasseParent):
    # Attributs et méthodes spécifiques à la classe enfant
```

Composition (Relation "A-un") :

```python
class ObjetContenu:
    def __init__(self, nom):
        self.nom = nom
    # Autres attributs et méthodes de la classe ObjetContenu

class ClasseContenant:
    def __init__(self):
        self.objet = ObjetContenu("nom_de_l_objet")
    # Autres attributs et méthodes de la classe ClasseContenant
```

Relations "Est-un" & "A-un"

Exemple

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

class Moteur:
    def __init__(self, type):
        self.type = type

class Voiture(Vehicule): # Héritage (est-un)
    def __init__(self, nom, moteur):
        super().__init__(nom)
        self.moteur = moteur # Composition (a-un)

    def afficher_info(self):
        print(self.nom + "a un moteur de type" + self.moteur.type)

moteur_voiture = Moteur("Essence")
ma_voiture = Voiture("Sedan", moteur_voiture)
ma_voiture.afficher_info()
```

Exercice 2

Soient les classes suivantes :

```python
class Véhicule: # classe de base
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee

class Voiture(Véhicule): # classe dérivée
    def __init__(self, marque, annee, modele):
        super().__init__(marque, année)
        self.modele = modele
```

1. Créez une nouvelle classe **MoteurElectrique** avec un attribut **batterie** pour représenter le type de batterie (par exemple, "lithium-ion").
2. Créez une sous-classe de Voiture nommée **VoitureElectrique**, qui inclue un attribut **moteur** de type MoteurElectrique.
3. Créez une instance de la classe *VoitureElectrique*.
   Afficher l'attribut batterie du moteur de la VoitureElectrique.

Exercice 2 I

Solution

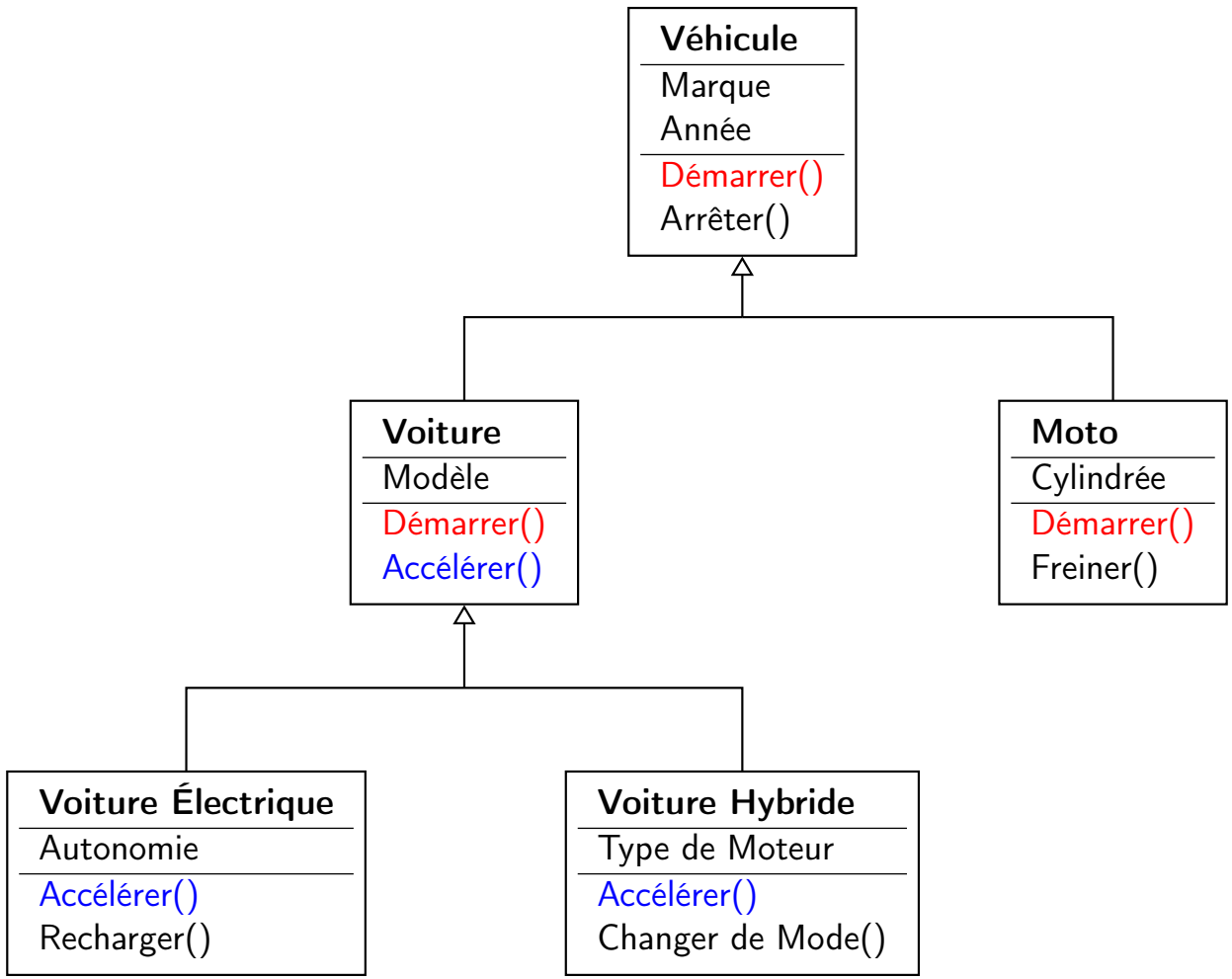
```python
class MoteurElectrique:
    def __init__(self, batterie):
        self.batterie = batterie

class VoitureElectrique(Voiture): # la relation est-un
    def __init__(self, marque, annee, modele, moteur):
        Voiture. __init__(self, marque, annee, modele)

        # la relation A-un
        assert isinstance(moteur, MoteurElectrique)
        self.moteur = moteur

# Création d'instances
moteurElect = MoteurElectrique("Lithium-ion")
voitureElect = VoitureElectrique("BMW", 2023, "X1", moteurElect)

# La batterie du(.) moteur de(.) la voitureElectrique
print(voitureElect.moteur.batterie)
```

Polymorphisme

Définition

Le **polymorphisme** se manifeste à travers l'implémentation de *méthodes avec le même nom dans différentes classes*.

Exemple :



Polymorphisme

Le polymorphisme permet à des objets de différentes classes de répondre à une même méthode ou opération de manière adaptée à leur propre classe.

- Si vous avez une méthode `démarrer()` dans les classes `Véhicule`, `Voiture`, et `Moto`, chaque classe peut implémenter cette méthode de manière différente.
- Appeler `démarrer()` sur une instance de `Voiture` affichera un message comme "La voiture démarre" tandis que pour une instance de `Moto`, cela pourrait afficher "Le moto démarre".

→ Le **Polymorphisme** permet d'utiliser une **interface uniforme** tout en conservant des **comportements spécifiques à chaque classe**.

Polymorphisme I

Exemple 1

```
1  class Vehicule: # Classe parente
2      def __init__(self, marque, annee):
3          self.marque = marque
4          self.annee = annee
5      def demarrer(self):
6          return "Le véhicule démarre"
7
8  class Voiture(Vehicule): # Classe enfant Voiture
9      def __init__(self, marque, annee, modele):
10         super().__init__(marque, annee)
11         self.modele = modele
12     def demarrer(self):
13         return "La voiture démarre avec une clé"
14
15 class Moto(Vehicule): # Classe enfant Moto
16     def __init__(self, marque, annee, cylindree):
17         super().__init__(marque, annee)
18         self.cylindree = cylindree
19     def demarrer(self):
20         return "La moto démarre en appuyant sur un bouton"
21
```

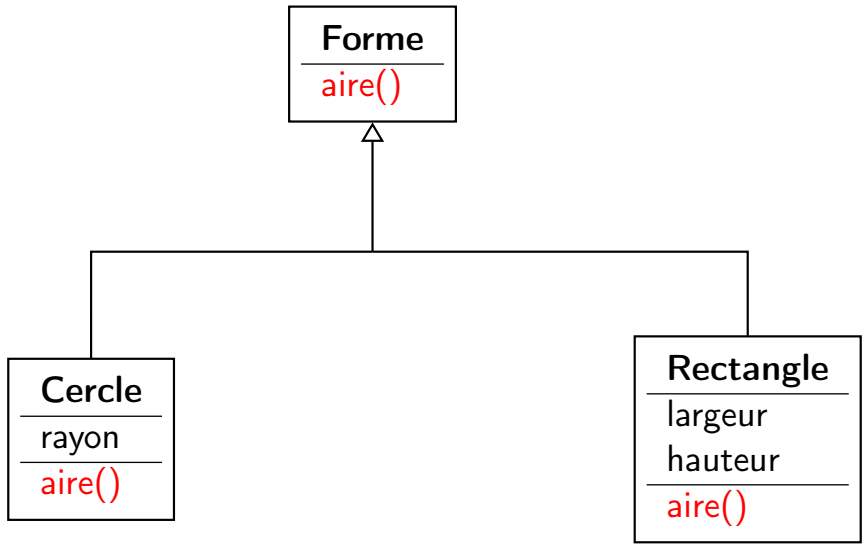
Polymorphisme II

Exemple 1

```
22      def freiner(self):
23          return "La moto freine"
24
25  # Création d'objets et démonstration du polymorphisme
26  # 1. Création d'objets
27  vehicules = [
28      Voiture("Toyota", 2020, "Corolla"),
29      Moto("Yamaha", 2019, "600cc")
30  ]
31
32  # 2. Utilisation polymorphique de la méthode `demarrer()`
33  for vehicule in vehicules:
34      # Affiche le message spécifique de démarrage selon le type de véhicule
35      print(vehicule.demarrer())
36
37
```

Polymorphisme

Exemple 2: Diagramme de classes



- Les classes Cercle et Rectangle héritent de la classe abstraite Forme, ce qui leur permet d'implémenter leur propre méthode aire(), illustrant ainsi le principe du polymorphisme où une méthode peut avoir différentes implémentations selon le type d'objet qui l'appelle.

Polymorphisme I

Exemple 2: Implémentation

```python
from math import pi
class Forme:
    def aire(self):
        pass # Traitement générique

class Cercle(Forme):
    def __init__(self, rayon):
        self.rayon = rayon
    def aire(self):
        return pi * (self.rayon ** 2)

class Rectangle(Forme):
    def __init__(self, largeur, hauteur):
        self.largeur = largeur
        self.hauteur = hauteur
    def aire(self):
        return self.largeur * self.hauteur

formes = [Cercle(5), Rectangle(4, 6)]
for forme in formes:
    print(f"Aire: {forme.aire()}")
```

Polymorphisme

Exercice 1

- Créez une classe `Polynome` avec une méthode `evaluer(x)` qui renvoie la valeur du polynôme pour une valeur donnée `x`.
- Créez ensuite des classes `PolynomeLineaire` (Degré 1, forme $ax + b$) et `PolynomeQuadratique` (Degré 2, forme $ax^2 + bx + c$) qui héritent de `Polynome`.
  Redéfinissez la méthode `evaluer(x)` dans chaque classe pour renvoyer les valeurs spécifiques des polynômes linéaires et quadratiques.

Polymorphisme I

Exercice 1: Solution

```
1  # Classe de base pour un polynôme
2  class Polynome: # classe abstraite
3      def evaluer(self, x):
4          raise NotImplementedError("La méthode 'evaluer' non implémentée")
5
6  # Classe pour un polynôme linéaire
7  class PolynomeLineaire(Polynome):
8      def __init__(self, a, b):
9          self.a = a
10         self.b = b
11
12     def evaluer(self, x):
13         return self.a * x + self.b
14
15 # Classe pour un polynôme quadratique
16 class PolynomeQuadratique(Polynome):
17     def __init__(self, a, b, c):
18         self.a = a
19         self.b = b
20         self.c = c
21
```

Polymorphisme II

Exercice 1: Solution

```
22      def evaluer(self, x):
23          return self.a * x**2 + self.b * x + self.c
24
25  # Tester le polymorphisme
26  # 1. Création d'objets
27  polynomes = [
28      PolynomeLineaire(2, 3),         # Représente 2x + 3
29      PolynomeQuadratique(1, -2, 1)  # Représente x^2 - 2x + 1
30  ]
31
32  # 2. Évaluation des polynômes pour x = 5
33  for polynome in polynomes:
34      print(f"Valeur du polynôme pour x = 5 : {polynome.evaluer(5)}")
35
```

À retenir

Héritage

- L'héritage est un mécanisme en POO qui permet de créer de nouvelles classes (enfants) basées sur des classes existantes (parents).
- Les classes enfants héritent des attributs et des méthodes des classes parentes.
- Cela favorise la réutilisation du code et la structuration des classes en hiérarchies.

Polymorphisme

- Le polymorphisme est la capacité à traiter des objets de différentes classes de manière uniforme.
- Souvent implémenté via la **surcharge** des méthodes (rédéfinir une méthode de la classe mère dans la classe fille).
- Il permet de réutiliser des méthodes ou des opérateurs sur des objets de différentes classes, simplifiant ainsi le code.