

# Héritage et composition

---

## Introduction

L'héritage et la composition constituent deux mécanismes fondamentaux de la conception orientée objet. Ils permettent d'organiser les relations entre les classes selon des logiques différentes mais complémentaires.

L'héritage traduit une relation hiérarchique de type « **est un** » (**is a**). Une classe dérivée hérite des attributs et des méthodes d'une classe parente, ce qui favorise la réutilisation du code et la spécialisation progressive du comportement. Lorsqu'on crée une sous-classe, on ne part pas de zéro : on étend ou on redéfinit certaines fonctionnalités de la classe de base afin d'adapter son comportement à un contexte plus précis. Ainsi, si une classe **Student** hérite de **Person**, elle dispose naturellement des propriétés et des méthodes de **Person**, tout en pouvant ajouter ses propres particularités. Ce mécanisme instaure une hiérarchie conceptuelle entre les entités ; il exprime une forme de **généralisation/spécialisation**.

La composition, quant à elle, traduit une relation de type « **a un** » (**has a**). Plutôt que d'hériter d'une autre classe, un objet contient une ou plusieurs instances d'autres objets pour bénéficier de leurs fonctionnalités. Ce principe repose sur la **délégation** : au lieu d'étendre une classe existante, on l'utilise comme composant interne. Par exemple, une classe **Car** pourrait posséder un objet **Engine** ou **Wheel**, sans pour autant hériter de ces classes. Cette approche favorise la flexibilité et l'indépendance des composants ; elle évite les liens hiérarchiques rigides et facilite la maintenance.

L'héritage tend à établir des dépendances verticales et conceptuelles entre types apparentés, tandis que la composition privilégie des associations horizontales entre objets coopérant. Dans une architecture équilibrée, on privilégie souvent la composition à l'héritage dès lors qu'il s'agit de **partager des comportements** plutôt que de **représenter une identité commune**, suivant ainsi le principe de conception : « *Favoriser la composition plutôt que l'héritage* ».

## L'héritage

L'héritage **modélise une relation "est un" (is a ...)**. On définit une classe de base **Person** dont **Student** hérite.

```
class Person:
    def __init__(self, prenom: str, nom: str, age: int) -> None:
        self.prenom = prenom
        self.nom = nom
        self.age = age

    def dire_bonjour(self) -> str:
        return "Bonjour tout le monde !"

class Student(Person):
    def __init__(self, prenom: str, nom: str, age: int, matiere: str) -> None:
        super().__init__(prenom, nom, age)
        self.matiere = matiere
```

```
# Démonstration
alice = Student("Alice", "Smith", 19, "Finance")
print(alice.dire_bonjour())          # "Bonjour tout le monde !" (hérite de
Person)
print(isinstance(alice, Student))    # True
print(isinstance(alice, Person))     # True
```

On définit une Person par son prenom, son nom et son âge. On définit également une méthode `dire_bonjour` qui renvoie une chaîne de caractères.

Student hérite de Person par l'écriture :

```
class Student(Person)
```

En effet, Student est une Person, donc il hérite de Person.

L'héritage confert à Student la méthode `dire_bonjour` de Person.

Le mot-clé `super()` en Python est un mécanisme qui permet à une classe dérivée de **communiquer avec sa classe parente**. Il sert à invoquer explicitement une méthode héritée du parent depuis la sous-classe, tout en respectant la hiérarchie d'héritage.

Dans l'exemple suivant :

```
class Student(Person):
    def __init__(self, prenom: str, nom: str, age: int, matiere: str) ->
None:
    super().__init__(prenom, nom, age)
    self.matiere = matiere
```

la méthode `__init__` de `Student` redéfinit celle de `Person` pour ajouter un attribut supplémentaire `matiere`.

Cependant, avant de s'occuper de sa propre initialisation, `Student` appelle `super().__init__(prenom, nom, age)` afin d'exécuter l'initialisation prévue dans la classe parente `Person`.

Cela permet de ne pas dupliquer le code qui configure déjà les attributs `prenom`, `nom` et `age`.

On peut considérer `super()` comme **une interface de communication ascendante** : il établit un lien contrôlé entre l'enfant et son parent. Ce lien assure la continuité du processus d'initialisation ou d'exécution, en appelant la méthode parente correspondante dans l'ordre de résolution des méthodes (appelé MRO, *Method Resolution Order*).

Ainsi, grâce à `super()`, la sous-classe profite des comportements hérités tout en les complétant. Dans ce cas précis, `Person` se charge d'initialiser les données de base, tandis que `Student` se concentre sur les éléments spécifiques à son rôle. Cela montre parfaitement la collaboration entre les niveaux de la hiérarchie

d'héritage : l'enfant s'appuie sur le parent pour prolonger et spécialiser son comportement, sans pour autant le remplacer entièrement.

Étape 1, **Student** ne redéfinit rien et utilise la méthode du parent telle quelle. On observe la délégation implicite vers **Person** grâce à la résolution de méthodes. Ici, **Student** ne possède pas **dire\_bonjour**, donc l'appel remonte vers **Person** dans l'ordre de résolution (MRO), et on récupère exactement la même chaîne.

Étape 2, **Student** redéfinit **dire\_bonjour**. On introduit un **masquage** : la méthode homonyme définie dans **Student** "cache" celle de **Person** pour les instances **Student**.

```
class Student(Person):
    def __init__(self, prenom: str, nom: str, age: int, matiere: str) ->
None:
        super().__init__(prenom, nom, age)
        self.matiere = matiere

    # Redéfinition sans appel au parent
    def dire_bonjour(self) -> str:
        return f"Bonsoir, je suis {self.prenom} et j'étudie la
{self.matiere}."

# Démonstration
alice = Student("Alice", "Smith", 19, "Finance")
print(alice.dire_bonjour())      # Bonjour, je suis Alice et j'étudie la
Finance.
print(isinstance(alice, Student)) # True
print(isinstance(alice, Person))  # True
```

Étape 3, Dans cette version, **Student** fait appel au comportement de son parent grâce à **super()**. On commence par récupérer la salutation "générique" de **Person**, puis on la modifie avant de la renvoyer. Cela permet de garder le message d'origine tout en l'adaptant au contexte de l'étudiant. Par exemple, on transforme "Bonjour" en "Bonsoir" et on y ajoute des informations propres à **Student**.

```
class Student(Person):
    def __init__(self, prenom: str, nom: str, age: int, matiere: str) ->
None:
        super().__init__(prenom, nom, age)
        self.matiere = matiere

    # Redéfinition avec appel explicite au parent
    def dire_bonjour(self) -> str:
        base = super().dire_bonjour() # "Bonjour tout le monde !"
        base_personnalisee = base.replace("Bonjour", "Bonsoir")
        return f"{base_personnalisee} Je suis {self.prenom} en
{self.matiere}. Comment ça va ?"

# Démonstration
alice = Student("Alice", "Smith", 19, "Finance")
print(alice.dire_bonjour())      # Bonsoir, je suis Alice en Finance.
```

Comment ça va ?

```
print(isinstance(alice, Student)) # True
print(isinstance(alice, Person))  # True
```

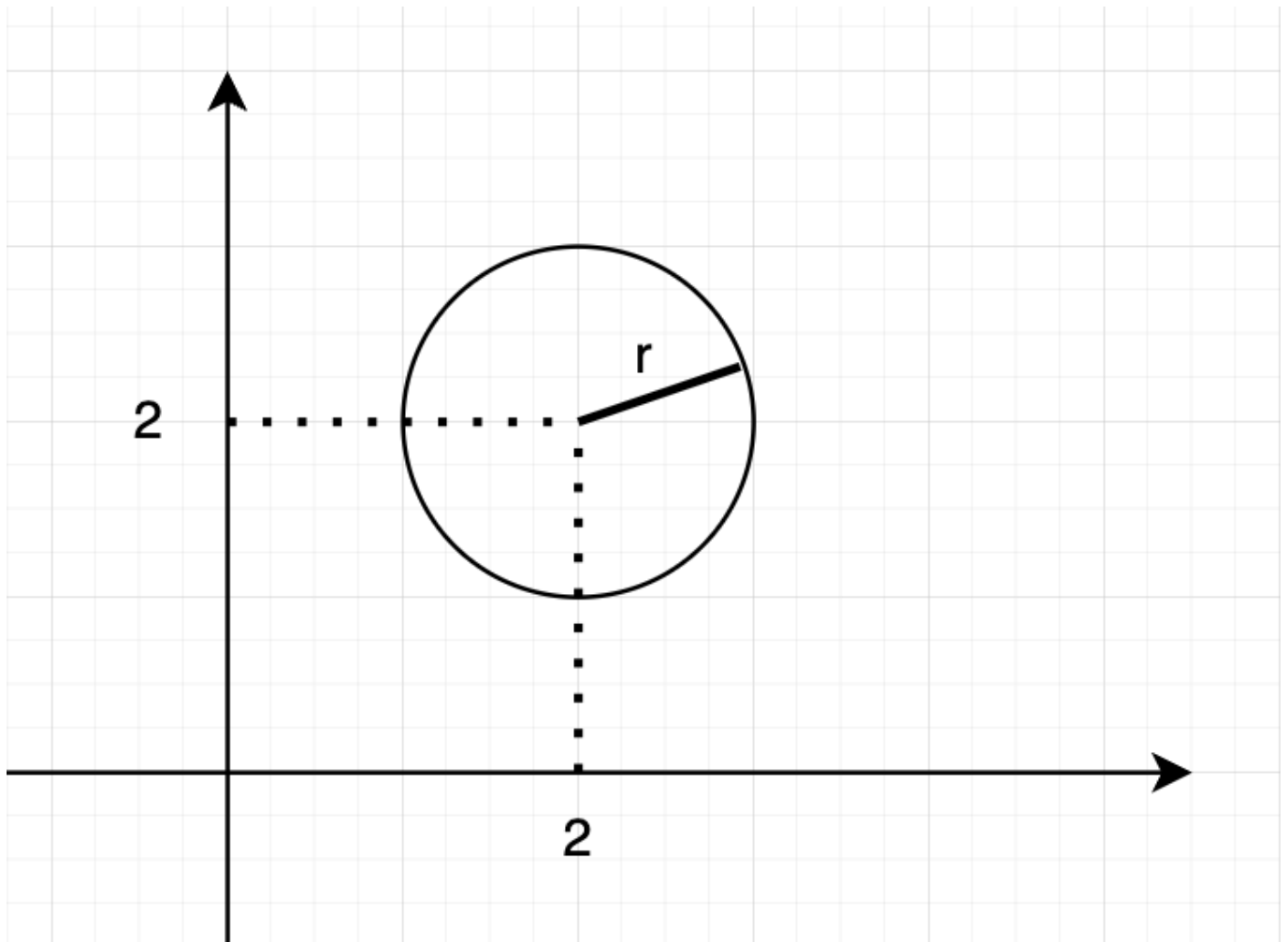
On retient que Student “est un(e)” Person, ce que confirment les tests **isinstance**, que la résolution de méthodes appelle d’abord la version la plus spécialisée disponible dans la hiérarchie, et que `super()` permet à une sous-classe d’étendre proprement un comportement parent plutôt que de le réécrire entièrement. Ce schéma respecte l’esprit de la spécialisation : on préserve la sémantique de Person, puis on l’adapte au contexte de Student sans rompre le contrat.

Retenez qu’à chaque fois qu’une classe fille voudra récupérer un attribut, ou une méthode, elle fera appel via **super()** à la méthode de son parent.

## La composition

La **composition** décrit une relation où un objet **possède** un autre objet ou **dépend** de lui pour exister. On parle alors d’une relation de type « **a un** » (**has a**), contrairement à l’héritage qui traduit une relation « **est un** » (**is a**).

On souhaite par exemple définir une classe **Cercle** qui possède un attribut **centre** de type **Point** et un attribut **rayon** de type **float**.



On ne dit pas qu’un **Cercle EST UN Point**, mais qu’un **Cercle A UN Point** comme centre. Le centre n’existe pas indépendamment du cercle dans ce contexte : il est une partie constitutive de celui-ci.

Le Point peut être représenté par une classe `Point` et le Cercle par une classe `Cercle` définie comme suit :

```
class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

class Cercle:
    def __init__(self, centre: Point, rayon: float):
        self.centre = centre
        self.rayon = rayon

cercle = Cercle(
    centre=Point(2, 2),
    rayon=5
)
```

Le typage ici joue un rôle essentiel pour comprendre la nature de la composition.

Dans la signature du `__init__` de `Cercle`, le paramètre `centre` est annoté avec le type `Point`. Cela indique clairement que l'attribut `centre` doit être une instance de la classe `Point`. Grâce à ce typage, on comprend immédiatement que `Cercle` dépend d'un autre objet de type `Point` pour être construit.

Du point de vue du **Cercle**, on parle du "centre du cercle", c'est une relation de possession : le centre est une propriété interne du cercle. Mais d'un point de vue du **type**, ce centre **est un objet de type Point**, ce qui montre qu'il s'agit d'une instance réutilisable d'une autre classe.

Ainsi, la composition permet de **structurer un objet complexe à partir d'objets plus simples**, tout en gardant des relations claires entre les types. Elle favorise la modularité : on peut réutiliser la classe `Point` dans d'autres contextes (par exemple pour un rectangle ou un triangle), sans modifier la définition du `Cercle`.

Dans cet exemple, on remarque que la classe `Cercle` **ne possède pas directement** les coordonnées `x` et `y`. Ces deux valeurs appartiennent à l'objet `centre`, qui est lui-même une instance de la classe `Point`.

Cela signifie que, pour accéder à la valeur de `x` du cercle, on doit **d'abord entrer dans l'objet centre**, car c'est lui qui contient les coordonnées. Le cercle ne fait que **contenir** ce point, il ne le remplace pas.

On peut illustrer cela ainsi :

```
print(cercle.centre.x)
```

Ici, l'interpréteur suit une chaîne d'accès logique :

1. `cercle` représente l'objet de type `Cercle`.
2. `cercle.centre` désigne l'attribut `centre` du cercle, qui est un objet de type `Point`.
3. `cercle.centre.x` accède à la propriété `x` de cet objet `Point`.

Cette structure hiérarchique montre bien la nature de la **composition** : le **Cercle** **contient** un **Point** complet, et pour accéder à ses données internes, on doit traverser les niveaux d'objets.

Ainsi, si on affiche :

```
print(f"Coordonnée x du centre du cercle : {cercle.centre.x}")
print(f"Coordonnée y du centre du cercle : {cercle.centre.y}")
```

on obtient :

```
Coordonnée x du centre du cercle : 2
Coordonnée y du centre du cercle : 2
```

Cette façon d'accéder à une valeur illustre bien le principe de la composition : **un objet est composé d'autres objets**, et pour atteindre une information précise, on doit naviguer à travers cette composition.

La composition permet de concevoir des objets plus petits, plus clairs et plus faciles à maintenir. L'idée est de **diviser un objet complexe en plusieurs sous-objets spécialisés**, chacun responsable d'un aspect précis du domaine. On ne regroupe pas toutes les données dans une seule classe volumineuse, mais on assemble plusieurs entités cohérentes entre elles.

Prenons l'exemple d'un utilisateur. Si l'on regroupait toutes ses informations dans une seule classe **User**, on obtiendrait une structure encombrée contenant des dizaines d'attributs (**prenom**, **nom**, **age**, **rue**, **ville**, **cp**, **email**, **telephone**, etc.). Ce genre d'objet devient vite difficile à comprendre, à tester et à faire évoluer.

La composition permet d'éviter cela en séparant les responsabilités :

- La classe **Identity** s'occupe uniquement des informations personnelles de l'utilisateur.
- La classe **Adresse** décrit la localisation postale.
- La classe **Contact** gère les moyens de communication.

```
class Identity:
    def __init__(self, prenom: str, nom: str, age: int):
        self.prenom = prenom
        self.nom = nom
        self.age = age

class Adresse:
    def __init__(self, rue: str, cp: str, ville: str):
        self.rue = rue
        self.cp = cp
        self.ville = ville

class Contact:
    def __init__(self, email: str, telephone: str):
        self.email = email
```

```
        self.telephone = telephone

class User:
    def __init__(self, identity: Identity, adresse: Adresse, contact:
Contact):
        self.identity = identity
        self.adresse = adresse
        self.contact = contact
```

L'objet **User** n'a donc que trois attributs, chacun représentant une entité à part entière, mais assemblée pour former un tout cohérent. Ce découpage rend la classe **User** légère, tout en permettant aux classes **Identity**, **Adresse** et **Contact** d'exister indépendamment dans d'autres contextes.

Lorsqu'on crée un utilisateur, on combine ces objets comme des briques :

```
john = User(
    identity=Identity('John', 'Doe', 34),
    adresse=Adresse('25 rue de la paix', '59000', 'Lille'),
    contact=Contact(email='john@gmail.com', telephone='0102030405'),
)
```

Grâce à ce découpage, **User** n'a pas besoin de connaître les détails internes de chaque sous-objet. Il sait seulement qu'il possède une **identité**, une **adresse** et un **contact**. Si demain on change la structure de **Adresse** ou qu'on ajoute de la logique dans **Contact**, la classe **User** n'aura pas besoin d'être modifiée.

Certains grands noms dans le domaine IT, comme **Yegor Bugayenko** (auteur du livre "**Elegant Object**"), vont même jusqu'à affirmer qu'un objet ne devrait pas avoir plus de **trois attributs**. Cette règle vise à limiter la complexité cognitive : au-delà de trois, un objet commence à mélanger plusieurs responsabilités. En composant les objets plutôt qu'en les gonflant, on obtient des classes plus expressives, plus faciles à tester et plus résistantes aux changements.

Ainsi, la composition n'est pas qu'une technique de structuration : c'est une philosophie qui pousse à concevoir des objets **simples, autonomes et réutilisables**, capables de collaborer sans dépendances excessives.

## Marier Composition et héritage

On marie héritage et composition en distinguant l'identité commune des formes géométriques de leurs composants internes.

```
from math import pi

class Point:
    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

class Forme:
```

```
def __init__(self, centre: Point) -> None:
    self.centre = centre

def aire(self) -> float:
    raise NotImplementedError

def perimetre(self) -> float:
    raise NotImplementedError
```

On commence par un objet élémentaire `Point`, puis on définit une classe abstraite de fait `Forme` qui possède un centre. Le centre est une composition : il s'agit d'un attribut de type `Point`, dont on dépend pour positionner la forme dans le plan. Ce choix est explicite grâce aux annotations de types, qui rendent visible l'intention "has a" dans la signature du constructeur.

Dans la classe `Forme`, on utilise l'instruction `raise NotImplementedError` dans les méthodes `aire()` et `perimetre()` pour imposer aux classes filles de les redéfinir.

Ici, on ne signale pas une erreur accidentelle, mais on crée volontairement **un point d'obligation** dans la hiérarchie d'héritage.

On indique ainsi que ces méthodes doivent être implémentées par toute classe qui hérite de `Forme`.

```
class Cercle(Forme):
    def __init__(self, centre: Point, rayon: float) -> None:
        super().__init__(centre)
        self.rayon = rayon

    def aire(self) -> float:
        return round(pi * self.rayon ** 2, 2)

    def perimetre(self) -> float:
        return round(2 * pi * self.rayon, 2)
```

On spécialise ensuite `Forme` par héritage avec `Cercle`. On réutilise l'initialisation du parent via `super().__init__(centre)` pour conserver la logique de positionnement, puis on ajoute l'attribut spécifique `rayon`. Les méthodes `aire` et `perimetre` concrétisent le contrat de `Forme` pour ce sous-type.

L'héritage exprime ici la relation "est une forme", tandis que la présence d'un `Point` comme centre continue d'exprimer la composition.

On applique la même logique à `Rectangle`. On garde la même identité abstraite de "forme" par héritage, on conserve la composition par le `centre: Point`, et on introduit les dimensions propres à un rectangle. L'interface commune (`aire`, `perimetre`) rend polymorphes les sous-classes, ce qui autorise un traitement uniforme des formes.

```
class Rectangle(Forme):
    def __init__(self, centre: Point, longueur: float, largeur: float) ->
None:
```



```

    super().__init__(centre)
    self.longueur = longueur
    self.largeur = largeur

    def aire(self) -> float:
        return self.longueur * self.largeur

    def perimetre(self) -> float:
        return 2 * (self.longueur + self.largeur)

```

On peut alors manipuler ces objets de façon homogène et naviguer dans la composition pour accéder aux coordonnées. On accède à `x` et `y` en passant par l'attribut `centre`, car ce dernier est de type `Point` et encapsule les coordonnées.

```

p: Point = Point(2.0, 2.0)
c: Cercle = Cercle(centre=p, rayon=5.0)
r: Rectangle = Rectangle(centre=Point(0.0, 1.0), longueur=4.0,
largeur=3.0)

print(c.aire(), c.perimetre(), c.centre.x, c.centre.y)      # 78.54 31.42
2.0 2.0
print(r.aire(), r.perimetre(), r.centre.x, r.centre.y)      # 12.0 14.0 0.0
1.0

```

On ne considère pas qu'un paradigme soit plus "fort" qu'un autre.

On choisit l'héritage lorsqu'on veut exprimer une identité commune et un contrat stable à respecter dans une hiérarchie de types, afin de bénéficier du polymorphisme et de la spécialisation contrôlée.

On choisit la composition lorsqu'on veut construire des objets par assemblage de composants autonomes et réutilisables, en gardant un couplage plus souple.

Dans ce modèle, on exploite l'héritage pour le **quoi** (être une `Forme` avec une interface `aire/perimetre`) et la composition pour le **avec quoi** (posséder un `Point` comme centre), ce qui permet d'évoluer sereinement avec des responsabilités bien séparées et lisibles par le typage.

## Héritage et Composition dans une Application de Gestion Professionnelle

Dans un contexte professionnel, on rencontre très souvent la combinaison de l'**héritage** et de la **composition** dans les logiciels de gestion, qu'il s'agisse de gérer des employés, des clients, des produits ou des commandes.

Prenons l'exemple d'un **système de gestion du personnel** dans une entreprise. On veut représenter différents types d'employés, chacun ayant des caractéristiques communes mais aussi des particularités propres à son poste.

On peut commencer par une classe `Employe`, qui contient les informations communes à tous les employés : identité, salaire, et méthode de calcul du revenu mensuel. Cette classe jouera le rôle de **classe de base**.

```
class Identity:
    def __init__(self, prenom: str, nom: str):
        self.prenom = prenom
        self.nom = nom

class Employe:
    def __init__(self, identity: Identity, salaire: float):
        self.identity = identity
        self.salaire = salaire

    def revenu_mensuel(self) -> float:
        return self.salaire
```

Ici, la classe **Employe** utilise la **composition** : elle possède un objet **Identity**. Cela évite de dupliquer les attributs **prenom** et **nom** partout, et rend le code plus clair.

Ensuite, on peut spécialiser certains employés grâce à l'**héritage**. Par exemple, un **Commercial** hérite d'**Employe**, mais ajoute une logique spécifique : il reçoit des commissions sur les ventes.

```
class Commercial(Employe):
    def __init__(self, identity: Identity, salaire: float, commission: float):
        super().__init__(identity, salaire)
        self.commission = commission

    def revenu_mensuel(self) -> float:
        return self.salaire + self.commission
```

De la même manière, un **Developpeur** pourrait hériter d'**Employe** mais avoir un mode de rémunération différent, par exemple un bonus basé sur les projets terminés.

```
class Developpeur(Employe):
    def __init__(self, identity: Identity, salaire: float, bonus_projet: float):
        super().__init__(identity, salaire)
        self.bonus_projet = bonus_projet

    def revenu_mensuel(self) -> float:
        return self.salaire + self.bonus_projet
```

On peut ensuite créer et manipuler ces objets de manière polymorphe :

```
john = Commercial(Identity("John", "Doe"), salaire=2500, commission=500)
emma = Developpeur(Identity("Emma", "Smith"), salaire=3000,
bonus_projet=800)
```

```
employees = [john, emma]

for e in employees:
    print(f"{e.identity.prenom} gagne {e.revenu_mensuel()} € ce mois-ci.")
```

Sortie :

```
John gagne 3000.0 € ce mois-ci.
Emma gagne 3800.0 € ce mois-ci.
```

Dans cet exemple,

- **l'héritage** permet de définir une structure commune (**Employe**) et de spécialiser les comportements (**Commercial**, **Developpeur**) sans dupliquer de code.
- **la composition** permet de découper les données complexes (**Identity**) pour créer des objets plus simples, plus réutilisables et plus maintenables.

Dans une activité professionnelle, cette combinaison apparaît dès qu'on construit des modèles métier : par exemple dans les systèmes de facturation (Facture, LigneFacture, Produit), dans les ERP, ou dans les applications de gestion des ressources humaines. Elle traduit une manière naturelle d'organiser le code autour de concepts réels du domaine, tout en gardant une structure claire et évolutive.

## Le polymorphisme

Le **polymorphisme** est un concept central de la programmation orientée objet. Il vient du grec **polus** (plusieurs) et **morphê** (forme) et signifie littéralement "plusieurs formes". En pratique, cela veut dire qu'un même nom de méthode peut avoir plusieurs comportements différents, selon l'objet sur lequel elle est appelée.

### Sans polymorphisme

Prenons l'exemple d'un système de **notifications** dans une application. Chaque notification a la même intention, informer l'utilisateur, mais le moyen de communication diffère : certaines s'affichent à l'écran, d'autres s'envoient par email, d'autres encore par SMS.

Voici une première approche, sans polymorphisme :

```
class GestionnaireNotifications:
    def envoyer_email(self) -> str:
        return "Envoi d'un email à l'utilisateur."

    def envoyer_sms(self) -> str:
        return "Envoi d'un SMS sur le téléphone de l'utilisateur."

    def envoyer_push(self) -> str:
        return "Affichage d'une notification sur l'écran de l'utilisateur."
```

```
# Liste de notifications à traiter
notifications = [
    {"type": "email"},
    {"type": "sms"},
    {"type": "push"},
]

gestionnaire = GestionnaireNotifications()

# Il faut tester le type à chaque fois
for notif in notifications:
    if notif["type"] == "email":
        print(gestionnaire.envoyer_email())
    elif notif["type"] == "sms":
        print(gestionnaire.envoyer_sms())
    elif notif["type"] == "push":
        print(gestionnaire.envoyer_push())
```

### Problèmes de cette approche :

- Le code est rempli de conditions `if/elif`, ce qui le rend difficile à lire et à maintenir
- Si on veut ajouter une notification Slack, il faut modifier la classe ET ajouter un nouveau `elif` partout où on traite des notifications
- Chaque méthode a un nom différent (`envoyer_email`, `envoyer_sms`, etc.), ce qui casse la cohérence

### Avec polymorphisme

Avec le polymorphisme, on peut améliorer considérablement ce code :

```
class Notification:
    def envoyer(self) -> str:
        raise NotImplementedError("Méthode à redéfinir par les sous-classes")

class NotificationEmail(Notification):
    def envoyer(self) -> str:
        return "Envoi d'un email à l'utilisateur."

class NotificationSMS(Notification):
    def envoyer(self) -> str:
        return "Envoi d'un SMS sur le téléphone de l'utilisateur."

class NotificationPush(Notification):
    def envoyer(self) -> str:
        return "Affichage d'une notification sur l'écran de l'utilisateur."
```

Toutes ces classes héritent de **Notification**, mais redéfinissent la méthode **envoyer()** à leur manière. Le polymorphisme se manifeste lorsqu'on manipule plusieurs notifications sans connaître leur type précis :

```
notifications = [  
    NotificationEmail(),  
    NotificationSMS(),  
    NotificationPush(),  
]  
  
for n in notifications:  
    print(n.envoyer())
```

Sortie :

```
Envoi d'un email à l'utilisateur.  
Envoi d'un SMS sur le téléphone de l'utilisateur.  
Affichage d'une notification sur l'écran de l'utilisateur.
```

Le programme ne sait pas s'il s'agit d'un email, d'un SMS ou d'une notification visuelle. Il se contente d'appeler **envoyer()** sur chaque objet, et **Python choisit automatiquement la bonne version** selon la classe de l'objet.

Ce mécanisme présente trois avantages essentiels :

- Il **simplifie le code**, car on n'a pas besoin d'utiliser de conditions pour savoir quel type de notification traiter.
- Il **favorise l'extensibilité**, car on peut ajouter une nouvelle classe **NotificationSlack** sans modifier le reste du code.
- Il **renforce la cohérence** : toutes les notifications respectent le même contrat (la méthode **envoyer()**), tout en exprimant leur logique propre.

Le polymorphisme, c'est donc la capacité d'un même message (**envoyer**) à **prendre plusieurs formes** selon le contexte. C'est un des principes qui rend la programmation orientée objet à la fois élégante, flexible et évolutive.

## Conclusion

L'héritage et la composition forment deux piliers complémentaires de la programmation orientée objet. Ils permettent de structurer les programmes de manière claire, lisible et extensible, en traduisant les relations logiques entre les objets du monde réel.

Avec **l'héritage**, on décrit des relations hiérarchiques de type "est un". Une classe fille hérite des attributs et des comportements de sa classe mère, tout en pouvant les adapter ou les enrichir. C'est une approche qui favorise la spécialisation : on part d'un concept général pour créer des variantes plus précises, comme **Forme** → **Cercle** ou **Employe** → **Commercial**. L'héritage donne ainsi une cohérence aux objets d'un

même type et rend possible le polymorphisme, c'est-à-dire la possibilité de manipuler des objets différents à travers une même interface.

Avec **la composition**, on assemble des objets plus petits pour en construire de plus complexes. C'est une approche qui favorise la décomposition fonctionnelle : au lieu d'avoir de grandes classes contenant tout, on crée des objets spécialisés qui coopèrent. Cette modularité rend le code plus souple et plus facile à maintenir, car chaque objet garde une responsabilité limitée, comme **User** possédant une **Identity**, une **Adresse** et un **Contact**.

Ces deux approches ne s'opposent pas : elles se complètent. On utilise l'héritage pour partager des comportements et des structures communes, et la composition pour construire des ensembles cohérents à partir de composants réutilisables. Le bon équilibre dépend toujours du contexte. Dans un modèle métier, on privilégiera la composition pour découpler les entités ; dans une hiérarchie conceptuelle claire, on préférera l'héritage pour garantir une logique d'appartenance.

Maîtriser ces deux mécanismes, c'est apprendre à concevoir des objets simples, cohérents et collaboratifs, capables d'évoluer sans fragiliser le reste du système. C'est là le cœur de la pensée orientée objet : construire des logiciels qui modélisent le réel avec rigueur, clarté et durabilité.