

PYTHON

Maîtriser les fondamentaux de Python.

Utiliser ce langage pour aborder des problématiques avancées de programmation (performance, qualité du code)

**Cours – partie 1
Jean-Christophe RANGON
2024**



email: jc.rangon.formateur@gmail.com

linkedIn: www.linkedin.com/in/jean-christophe-rangon-dev-web

pour :



Table des matières

I. RAPPEL UML

II. CLASSES

III. HERITAGE

IV. ENCAPSULATION

V. MEMBRES STATIQUES

VI. CLASSES STATIQUES

VII. ABSTRACTION

VIII. INTERFACES

IX. TRAITS

X. PATRONS DE CONCEPTION

XI.TP

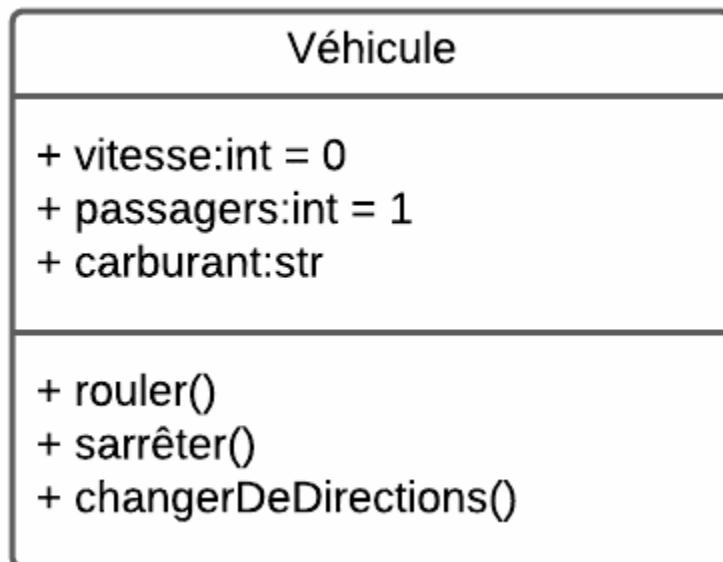
I. RAPPEL UML

UML (Unified Modeling Language) est un langage de modélisation standard utilisé pour visualiser, documenter et concevoir des systèmes logiciels complexes.

Les diagrammes UML les plus courants sont :

A. Les diagrammes de classe

Le diagramme de classe sert à décrire la structure statique d'un système en illustrant les classes, leurs attributs, leurs méthodes (ou opérations) et les relations entre elles.



Le diagramme de classes standard est composé de trois sections :

- **Section supérieure** : contient le nom de la classe.
Cette section est toujours nécessaire, que vous parliez du classifieur ou d'un objet.
- **Section intermédiaire** : contient les attributs de la classe. Utilisez-la pour décrire les qualités de la classe. Elle n'est nécessaire que lors de la description d'une instance spécifique d'une classe.
- **Section inférieure** : contient les opérations de la classe (méthodes), affichées sous forme de liste.
Chaque opération occupe sa propre ligne. Les opérations décrivent la manière dont une classe interagit avec les données.

a. Modificateurs d'accès des membres

Toutes les classes ont des niveaux d'accès différents, en fonction du modificateur d'accès (indicateur de visibilité). Voici les niveaux d'accès existants et les symboles qui leur sont associés :

Modificateurs d'accès

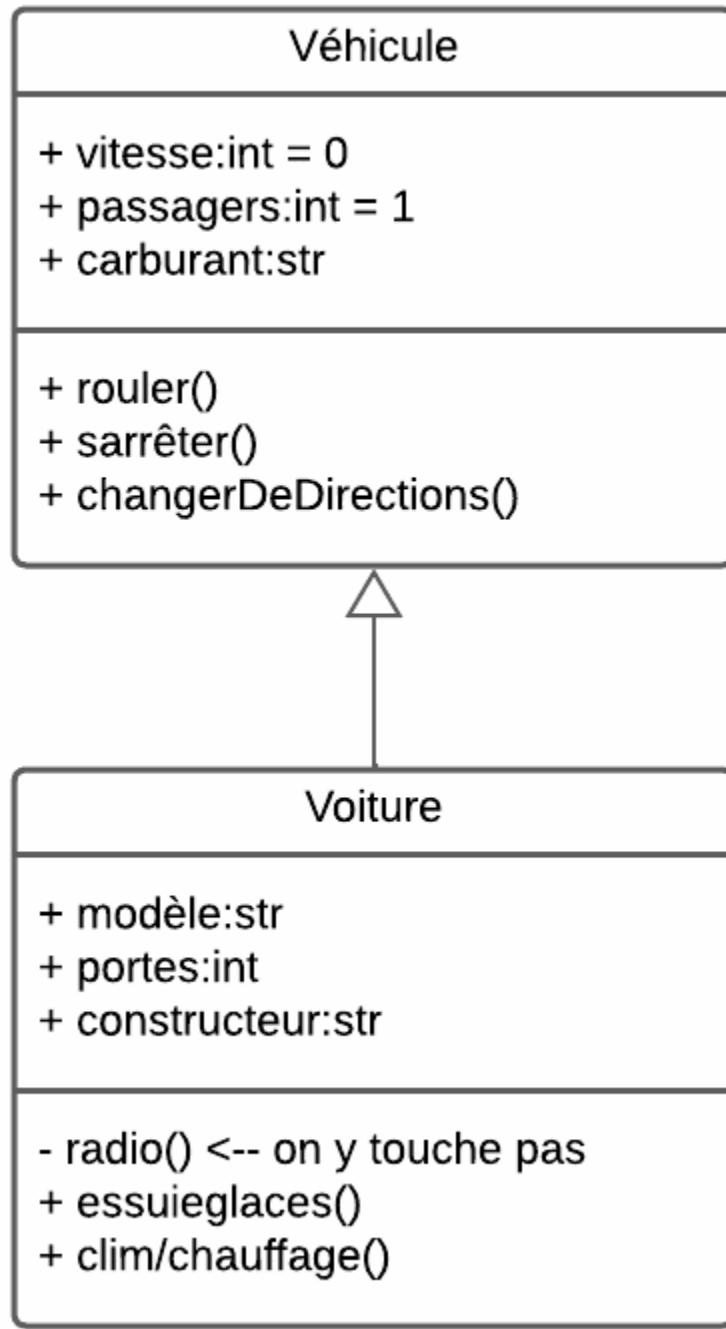
- Public (+)
- Privé (-)
- Protégé (#)
- Paquetage (~)
- Dérivé (/)
- Statique (souligné)

b. Interaction

Le terme « interactions » désigne les relations et liens divers qui peuvent exister dans les diagrammes de classes et d'objets. Voici quelques-unes des interactions les plus courantes :

i. Relation d'héritage

Également connu sous le nom de généralisation, il s'agit du processus par lequel un enfant ou une sous-classe adopte la fonctionnalité d'un parent ou d'une super-classe. On le symbolise par une ligne de connexion droite avec une pointe de flèche fermée orientée vers la super-classe.



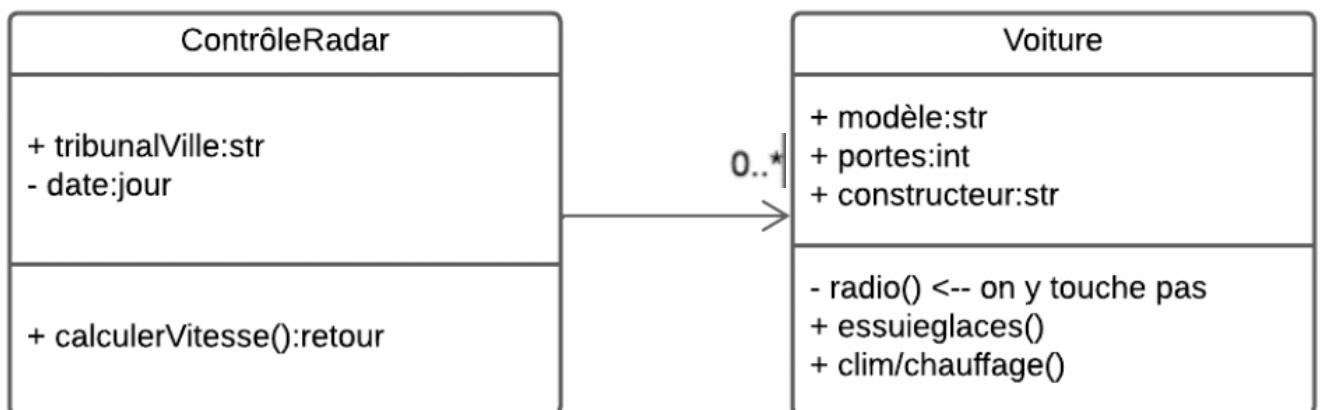
Dans cet exemple, l'objet « Voiture » hériterait de tous les attributs (vitesse, nombre de passagers, carburant) et méthodes (`rouler()`, `s'arrêter()`, `changerDeDirection()`) de la classe parent (« Véhicule »), en plus de ses

attributs spécifiques (type de modèle, nombre de portes, constructeur) et des méthodes de sa propre classe (Radio(), essuie-glace(), climatisation/chauffage()).

Dans un diagramme de classes, on représente l'héritage par une ligne pleine terminée par une flèche creuse et fermée.

ii. Relations d'association unidirectionnelle

Une **relation d'association unidirectionnelle** dans un diagramme de classes UML est une relation où une classe connaît ou interagit avec une autre classe, mais pas l'inverse. Autrement dit, seule une des deux classes est consciente de l'existence de l'autre. Cette relation reflète une dépendance où une classe utilise ou dépend des fonctionnalités d'une autre sans que cette dernière ait besoin de connaître la première.



Dans le diagramme ci-dessus, il y a une relation de dépendance de la classe « ControleRadar » vers la classe « Voiture ». En d'autre terme la classe ControleRadar a besoin d'une classe Voiture pour fonctionner. Mais la destruction d'une voiture n'entraîne pas l'arrêt des fonctionnalité de ControleRadar.

Si nous traduisons en pseudo-code :

Classe Voiture

Attributs :

- marque : String
- modèle : String
- vitesse : Integer

Méthodes :

- + Voiture(marque : String, modèle : String, vitesse : Integer) // constructeur
- + getMarque() : String
 - Retourne la marque de la voiture
- + setMarque(marque : String)
 - Définit la marque de la voiture
- + getModele() : String
 - Retourne le modèle de la voiture

```

+ setModele(modele : String)

    Définit le modèle de la voiture

+ getVitesse() : Integer

    Retourne la vitesse actuelle de la voiture

+ setVitesse(vitesse : Integer)

    Définit la vitesse de la voiture

+ accelerer(increment : Integer)

    // Augmente la vitesse de la voiture

    vitesse = vitesse + increment

+ freiner(decrement : Integer)

    // Diminue la vitesse de la voiture

    vitesse = vitesse - decrement

```

Fin Classe -----

Classe ControleRadar

Attributs :

- idRadar : Integer
- emplacement : String

Méthodes :

```

+ ControleRadar(idRadar : Integer, emplacement :
String)

+ getIdRadar() : Integer

```

```
        Retourne l'identifiant du radar  
+ setIdRadar(idRadar : Integer)  
        Définit l'identifiant du radar  
+ getEmplacement() : String  
        Retourne l'emplacement du radar  
+ setEmplacement(emplacement : String)  
        Définit l'emplacement du radar
```

```
+ detecterVitesse(voiture : Voiture)  
    // Déetecte si la vitesse de la voiture  
    dépasse la limite  
    Si voiture.getVitesse() > 100 Alors  
        enregistrerInfraction(voiture)  
    Fin Si
```

```
+ enregistrerInfraction(voiture : Voiture)  
    // Enregistre une infraction pour la voiture  
    détectée  
    Afficher "Infraction enregistrée pour la  
voiture : " + voiture.getMarque() + " " +  
voiture.getModele()
```

Fin Classe -----

Exemple d'exécution :

```
// Création d'une instance de Voiture
voiture1 = Voiture("Toyota", "Corolla", 120)

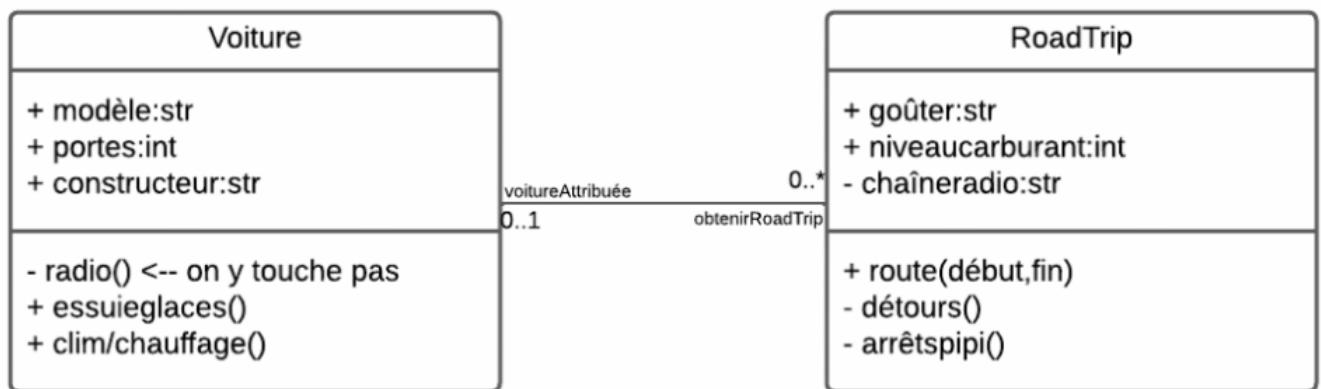
// Création d'une instance de ControleRadar
radar1 = ControleRadar(101, "Autoroute A1, km 45")

// Le radar détecte la vitesse de la voiture
radar1.detecterVitesse(voiture1)

// Sortie : "Infraction enregistrée pour la voiture :
Toyota Corolla"
```

iii. Relation d'association bidirectionnelle

C'est la relation par défaut entre deux classes. Chacune des deux classes a conscience de l'existence de l'autre et de sa relation avec elle. Cette association est représentée par une ligne droite entre deux classes. On y trouve les cardinalités



Dans l'exemple ci-dessus, la classe Voiture et la classe RoadTrip sont interdépendantes. À une extrémité de la ligne, la Voiture accepte l'association de « voitureAttribuée » avec la valeur de multiplicité de **0..1**, ce qui signifie que lorsque l'instance RoadTrip existe, elle peut être associée à 1 ou 0 instance de Voiture.

iv. Associations en détails

Class Diagram Relationship Type	Notation
Association	
Inheritance	
Realization/ Implementation	
Dependency	
Aggregation	
Composition	

v. Association de dépendance

Il est fondamental de comprendre la différence entre la relation d'association unidirectionnelle et la relation d'association de dépendance.

Prenons par exemple une classe Facture et une classe ServiceMail. La classe Facture a besoin de la classe ServiceMail pour envoyer une facture à un client.

Le pseudo-code donne ceci :

Classe ServiceEmail

Attributs :

- serveurSMTP : String
- port : Integer

Méthodes :

```
+ envoyerEmail(destinataire : String, message :  
String)
```

Fin Classe

Classe Facture

Attributs :

- numero : String
- montant : Double

Méthodes :

```
+ envoyerConfirmation()  
  
    serviceEmail = new ServiceEmail ("smtp.abc.com",  
587)  
  
    message = "Votre facture " + numero + " d'un  
montant de " + montant + "€ a été créée."  
  
    serviceEmail.envoyerEmail("client@example.com",  
message)
```

Fin Classe

Ici, la relation est fonctionnelle et temporaire, souvent limitée à l'exécution d'une méthode spécifique.

Une **Facture** utilise le **ServiceEmail** pour envoyer une confirmation. La classe **Facture** instancie la classe **ServiceEmail**.

Dans le cas de la Voiture et du **ContrôleRadar**, la relation est structurelle et nécessite une connaissance permanente de la cible. Un **ContrôleRadar** surveille plusieurs **Voitures**. La classe **ContrôleRadar** utilise des objets de classe Voiture mais n'instancie jamais une voiture.

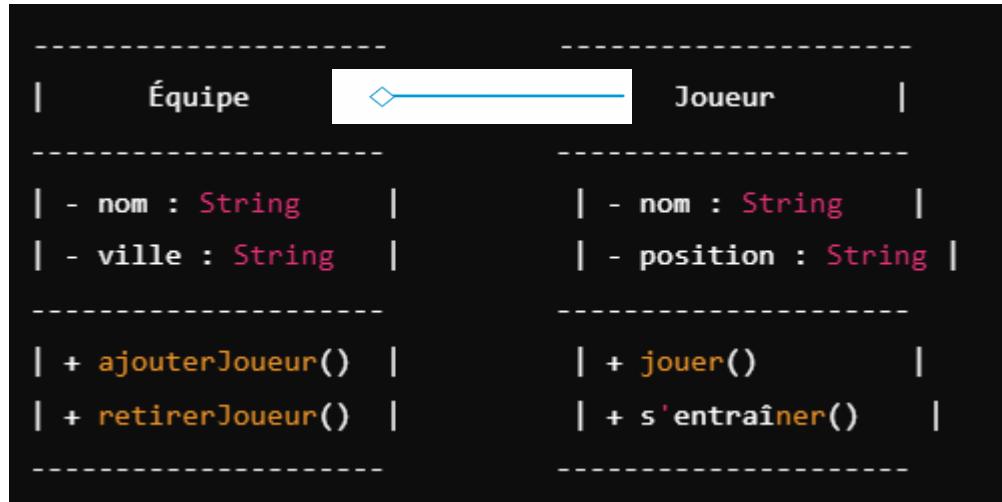
La relation de dépendance est symbolisée par :



vi. Association d'agrégation

L'**agrégation** est une forme spécialisée d'**association** dans les diagrammes de classes UML qui représente une relation **tout/partie** entre deux classes. Elle indique qu'une classe (le **tout**) est composée de une ou plusieurs instances d'une autre classe (les **parties**), mais

contrairement à l'association de composition, les parties peuvent exister indépendamment du tout.



Losange vide ($\langle \rangle$) : Indique une agrégation.

Direction : Le losange pointe vers **Equipe**, signifiant que **Equipe** agrège **Joueur**.

Multiplicité : Une **Équipe** peut avoir **0..*** **Joueurs**.

Examinons le pseudo code :

Classe Joueur

Attributs :

- nom : String
- position : String

Méthodes :

+ Joueur(nom : String, position : String)

// Constructeur

+ jouer()

Afficher nom + position

+ s_entraîner()

Afficher nom + " s'entraîne."

Fin Classe

Classe Équipe

Attributs :

- nom : String
- ville : String
- joueurs : **Tableau** de Joueurs

Méthodes :

```
+ Équipe(nom : String, ville : String)
    // Constructeur
    joueurs = Liste Vide
+ ajouterJoueur(joueur : Joueur)
    joueurs.Ajouter(joueur)
    Afficher joueur.nom + " a été ajouté à
l'équipe " + nom
+ retirerJoueur(joueur : Joueur)
    joueurs.Retirer(joueur)
    Afficher joueur.nom + " a été retiré de
l'équipe " + nom
+ jouerMatch()
    Afficher "L'équipe " + nom + " joue un
match."
Pour chaque joueur dans joueurs :
```

```
joueur.jouer()

+ entrainementEquipe()
    Afficher "L'équipe " + nom + " s'entraîne."
    Pour chaque joueur dans joueurs :
        joueur.s_entraîner()

Fin Classe
```

Exécution:

```
// Crédation de joueurs
joueur1 = Joueur("Alice", "Attaquant")
joueur2 = Joueur("Bob", "Défenseur")

// Crédation d'une équipe
équipe1 = Équipe("Les Lions", "Paris")

// Agrégation : L'équipe ajoute des joueurs
équipe1.ajouterJoueur(joueur1)
équipe1.ajouterJoueur(joueur2)

// L'équipe joue un match
équipe1.jouerMatch()

// Sortie :
// L'équipe Les Lions joue un match.
// Alice joue à la position Attaquant
```

```
// Bob joue à la position Défenseur  
  
// Un joueur est retiré de l'équipe  
equipe1.retirerJoueur(joueur1)  
  
// L'équipe s'entraîne  
  
equipe1.entrainementEquipe()  
  
// Sortie :  
  
// L'équipe Les Lions s'entraîne.  
  
// Bob s'entraîne.
```

Dans notre exemple,

- **Équipe** agrège **Joueur** :
- Équipe possède un tableau où elle stocke les Joueurs.
- Les Joueurs peuvent exister indépendamment de l'Équipe.

Par exemple, joueur1 pourrait rejoindre une autre équipe ou rester sans équipe.

L'**agrégation** est une relation puissante dans les diagrammes de classes UML qui permet de modéliser des relations **tout/partie** où les parties peuvent exister indépendamment du tout. Contrairement à la composition, l'agrégation ne lie pas la durée de vie des parties à

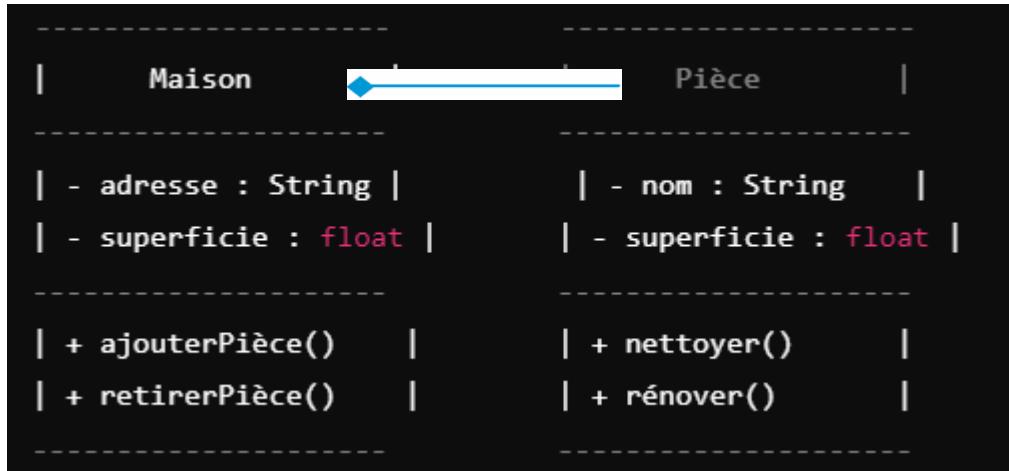
celle du tout, offrant ainsi une flexibilité accrue dans la conception des systèmes orientés objet.

Récapitulatif des Relations

Aspect	Association Unidirectionnelle	Association de Dépendance	Agrégation
Type de Relation	Structurelle, permanente	Fonctionnelle, temporaire	Tout/Partie, structurelle
Direction	Unidirectionnelle	Unidirectionnelle	Généralement bidirectionnelle
Durée de la Relation	Longue durée	Courte durée	Longue durée
Implication sur les Objets	Le tout connaît la partie	Le client utilise le fournisseur	Le tout agrège la partie
Représentation UML	Ligne solide avec flèche	Ligne en pointillés avec flèche	Ligne solide avec losange vide
Exemple Typique	ContrôleRadar connaît Voiture	Facture utilise ServiceEmail	Équipe agrège Joueur

vii. Association de Composition (agrégation forte)

La **composition** est une relation forte de **tout/partie** entre deux classes dans un diagramme de classes UML. Elle indique qu'une classe (**le tout**) est composée d'une ou plusieurs instances d'une autre classe (**les parties**), et que les parties ne peuvent pas exister indépendamment du tout. En d'autres termes, la durée de vie des parties est liée à celle du tout.



Dans un diagramme de classes UML, la **composition** est représentée par une **ligne solide avec un losange rempli** à l'extrémité du tout, pointant vers la partie.

Losange rempli (●) : Indique une composition.

Direction : Le losange pointe vers Pièce, signifiant que Maison est composée de Pièce.

Multiplicité : Une Maison peut avoir **1..*** Pièces.

Examinons le pseudo code suivant :

Classe Pièce

Attributs :

- nom : String
- superficie : Float

Méthodes :

```
+ Pièce(nom : String, superficie : Float)
    // Constructeur
+ nettoyer()
    Afficher "La pièce " + nom + " est en cours
de nettoyage."
+ rénover()
    Afficher "La pièce " + nom + " est en cours
de rénovation."
```

Fin Classe

Classe Maison

Attributs :

- adresse : String
- superficie : Float
- pièces : Liste de Pièce

Méthodes :

```
+ Maison(adresse : String, superficie : Float)

    // Constructeur

    pièces = Liste Vide

+ ajouterPièce(pièce : Pièce)

    pièces.Ajouter(pièce)

    Afficher "La pièce " + pièce.nom + " a été
ajoutée à la maison située à " + adresse

+ retirerPièce(pièce : Pièce)

    pièces.Retirer(pièce)

    Afficher "La pièce " + pièce.nom + " a été
retirée de la maison située à " + adresse

+ nettoyerMaison()

    Afficher "Nettoyage de la maison située à " +
adresse

Pour chaque pièce dans pièces :

    pièce.nettoyer()

+ rénoverMaison()

    Afficher "Rénovation de la maison située à "
+ adresse

Pour chaque pièce dans pièces :

    pièce.rénover()

Fin Classe
```

Exécution :

```
// Création de pièces
pièce1 = Pièce("Salon", 30.5)
pièce2 = Pièce("Cuisine", 20.0)

// Création d'une maison
maison1 = Maison("123 Rue Principale", 150.0)

// Composition : La maison ajoute des pièces
maison1.ajouterPièce(pièce1)
maison1.ajouterPièce(pièce2)

// La maison est nettoyée
maison1.nettoyerMaison()

// Sortie :
// Nettoyage de la maison située à 123 Rue Principale
// La pièce Salon est en cours de nettoyage.
// La pièce Cuisine est en cours de nettoyage.

// La maison est rénovée
maison1.rénoverMaison()

// Sortie :
```

```

// Rénovation de la maison située à 123 Rue Principale
// La pièce Salon est en cours de rénovation.
// La pièce Cuisine est en cours de rénovation.

// Suppression de la maison (si applicable, les pièces
sont également supprimées)

```

Dans notre exemple:

- **Maison** est le **tout** qui compose plusieurs **Pièces**.
- Les **Pièces** ne peuvent exister indépendamment de la **Maison**. Si la **Maison** est détruite, les **Pièces** le sont également.

Composition vs Agrégation

Aspect	Composition	Agrégation
Relation Tout/Partie	Relation forte : les parties ne peuvent pas exister sans le tout	Relation faible : les parties peuvent exister indépendamment du tout
Durée de Vie	Liée : destruction du tout entraîne la destruction des parties	Indépendante : destruction du tout ne détruit pas les parties
Représentation UML	Ligne solide avec losange rempli	Ligne solide avec losange vide
Exemple Typique	<code>Maison</code> composée de <code>Pièces</code>	<code>Équipe</code> agrège des <code>Joueurs</code>

viii. Associations de Réalisation/Implémentation

Réalisation

La **réalisation** est une relation UML utilisée principalement pour indiquer qu'une classe concrète **implémente** une interface. En d'autres termes, une classe qui réalise une interface fournit des implémentations concrètes pour les méthodes déclarées dans cette interface.



Exemple en pseudo-code :

Interface Printable

Méthodes :

```
+ imprimer() : Void
```

End Interface

Classe Document réalise Printable

Attributs :

```
- titre : String
```

```
- contenu : String
```

Méthodes :

```
+ Document(titre : String, contenu : String)
```

```
// Constructeur
```

```
this.titre = titre
```

```
this.contenu = contenu
```

```
+ imprimer()
```

```
Afficher "Impression du Document : " + titre
```

```
Afficher "Contenu : " + contenu
```

```
+ ajouterTexte(nouveauTexte : String)
```

```
contenu = contenu + "\n" + nouveauTexte
```

```
Afficher "Texte ajouté au Document : " + titre
```

End Classe

Exécution :

```
// Crédation d'un document
```

```
document1 = Document("Guide UML", "Introduction à UML.")

// Ajout de texte au document
document1.ajouterTexte("Section sur les diagrammes de
classes.")

// Impression du document
document1.imprimer()

// Sortie :
// Impression du Document : Guide UML
// Contenu : Introduction à UML.
// Section sur les diagrammes de classes.
```

Dans notre exemple :

Interface Printable :

- **Méthode imprimer()** : Déclare une méthode sans implémentation, destinée à être implémentée par les classes réalisant cette interface.

Classe Document :

- **Attributs** : titre et contenu pour stocker les informations du document.
- **Constructeur** : Initialise les attributs de Document.
- **Méthode imprimer()** : Fournit une implémentation concrète de la méthode déclarée dans Printable.
- **Méthode ajouterTexte()** : Ajoute du texte au contenu du document.

Document réalise l'interface **Printable** en fournissant une implémentation concrète de la méthode **imprimer()**.

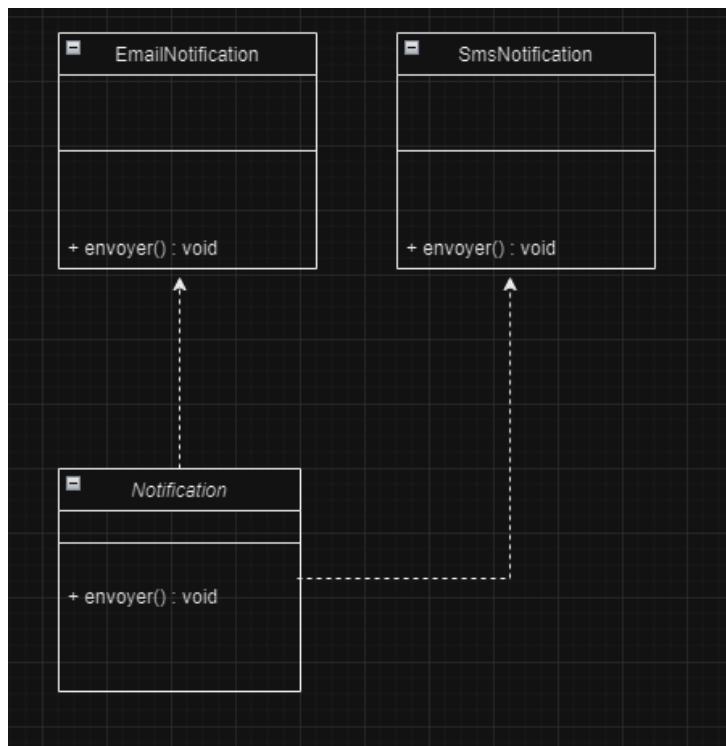
Printable définit le contrat que **Document** doit respecter, garantissant que toutes les classes réalisant **Printable** possèdent une méthode **imprimer()**.

Tableau récapitulatif :

Aspect	Association Unidirectionnelle	Association de Dépendance	Agrégation	Composition	Réalisation
Type de Relation	Structurelle, permanente	Fonctionnelle, temporaire	Tout/Partie, faible	Tout/Partie, forte	Interface/Class-concrète
Direction	Unidirectionnelle	Unidirectionnelle	Généralement bidirectionnelle	Généralement bidirectionnelle	Unidirectionnelle
Durée de la Relation	Longue durée	Courte durée	Longue durée	Liée à la durée de vie du tout	Longue durée
Implication sur les Objets	Le tout connaît la partie	Le client utilise le fournisseur	Le tout agrège la partie	Le tout possède et gère la partie	La classe implémente l'interface
Représentation UML	Ligne solide avec flèche	Ligne en pointillés avec flèche	Ligne solide avec losange vide	Ligne solide avec losange rempli	Ligne en pointillés avec triangle
Exemple Typique	ContrôleRadar connaît Voiture	Facture utilise ServiceEmail	Équipe agrège Joueurs	Maison composée de Pièces	Document réalise Printable

Exemple :

Imaginons un système de notification où plusieurs types de notifications (EmailNotification, SMSNotification) réalisent une interface Notification.



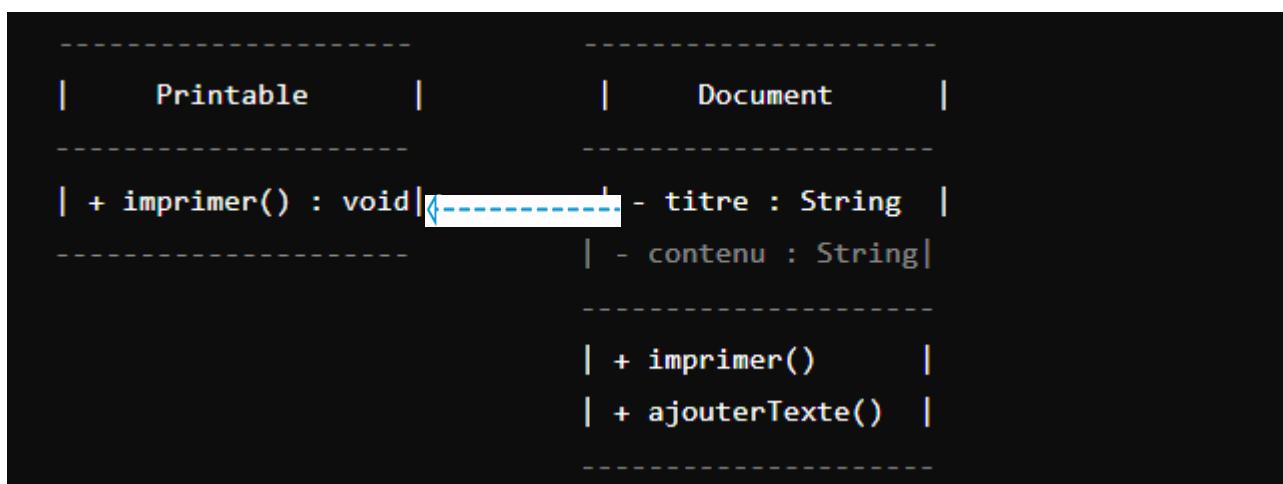
Notification est une interface définissant la méthode `envoyer()`.

EmailNotification, **SMSNotification** réalisent l'interface **Notification** en fournissant des implémentations concrètes de la méthode `envoyer()`.

Cela permet de traiter différentes notifications de manière uniforme via l'interface `Notification`.

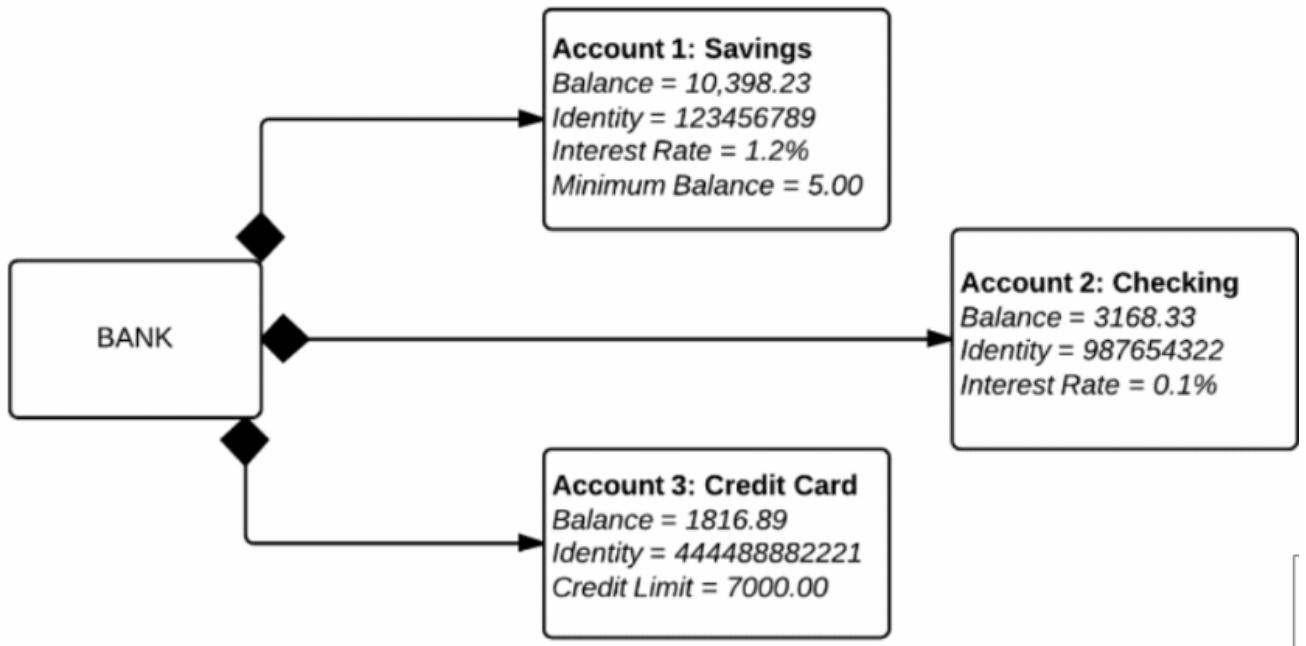
Implémentation

L'**implémentation** fait généralement référence au fait qu'une classe concrète fournit une définition concrète des méthodes déclarées dans une interface ou une classe abstraite. Dans UML, cette relation est représentée par une flèche en pointillés avec une pointe de flèche en forme de triangle creux pointant vers l'interface ou la classe abstraite.



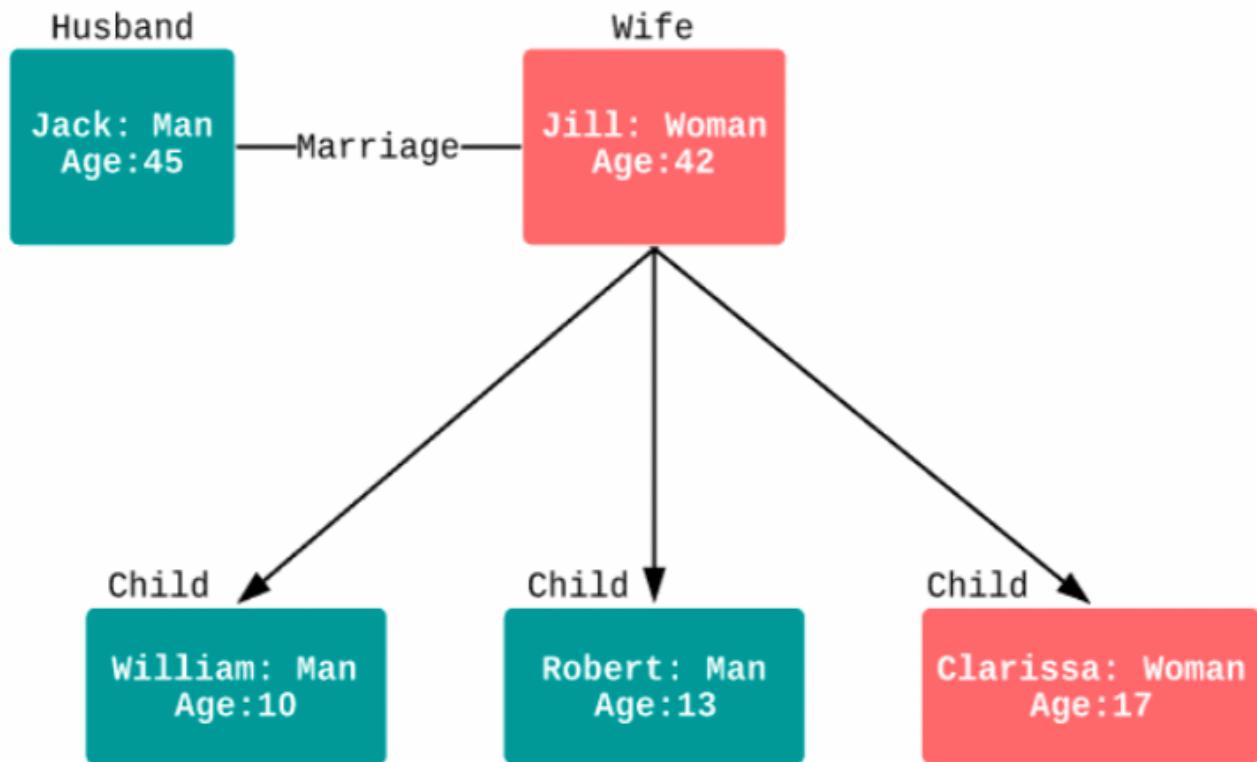
B. Les diagrammes d'objets

Un diagramme d'objet se concentre sur les attributs d'un ensemble d'objets et sur la façon dont ils interagissent les uns avec les autres.



Les trois comptes en banque sont reliés à la banque elle-même. Les titres de classes indiquent le type de compte (épargne, courant ou carte de crédit) qu'un client donné peut avoir dans cette banque. Les attributs de classes sont différents pour chaque type de compte. Ainsi, l'objet carte de crédit dispose d'une limite de crédit, alors que les comptes d'épargne et courant disposent de taux d'intérêt.

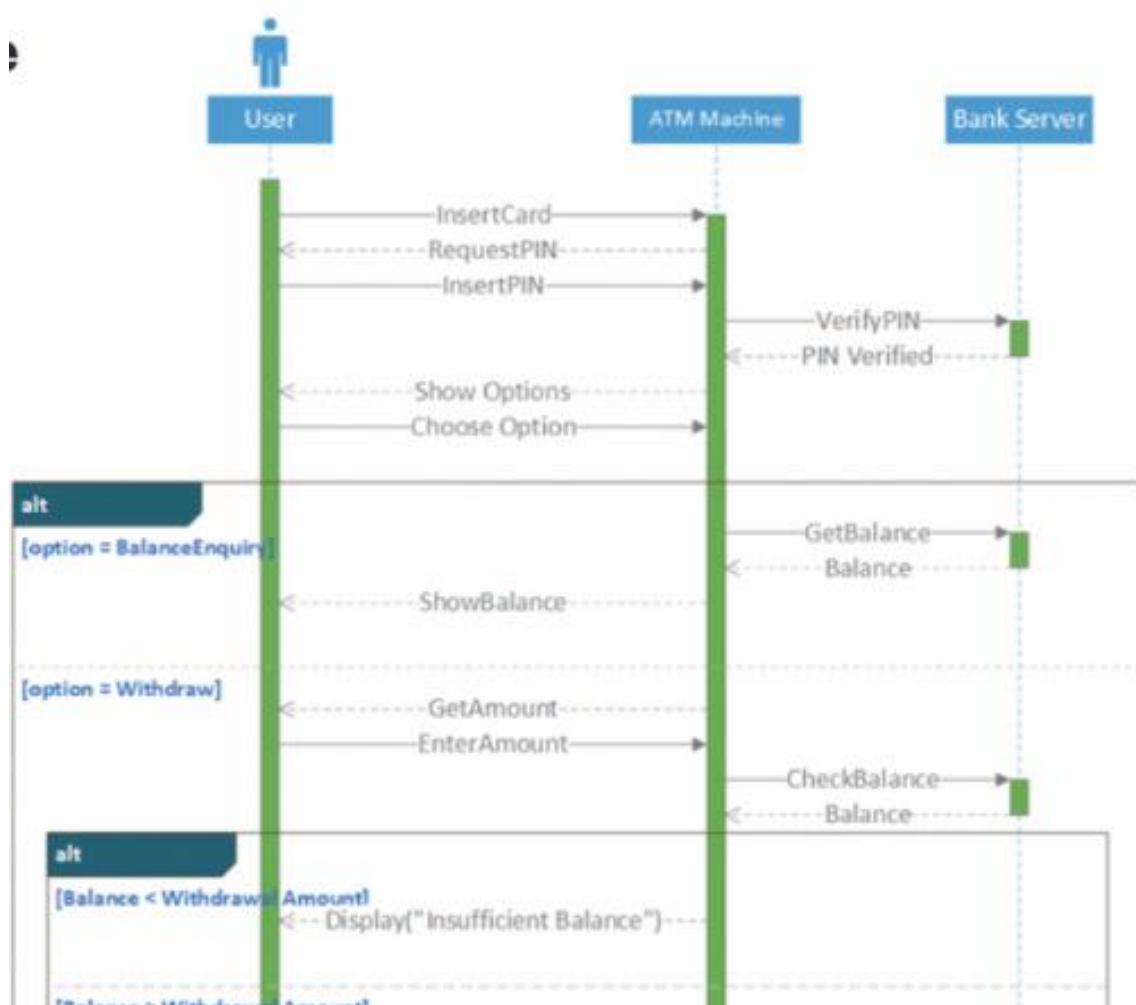
Exemple d'un arbre généalogique :



Dans le diagramme d'objet d'un arbre généalogique, ce sont le nom, le sexe et l'âge des membres de la famille - attributs de classe - qui sont importants. Ils peuvent apparaître en tant qu'éléments sur l'objet, voire dans les propriétés de l'objet même (par exemple la couleur)

C. Les diagrammes de séquence

Un diagramme de séquence est un type de diagramme d'interaction, car il décrit comment et dans quel ordre plusieurs objets fonctionnent ensemble. Ces diagrammes sont utilisés à la fois par les développeurs logiciels et les managers d'entreprises pour analyser les besoins d'un nouveau système ou documenter un processus existant. Les diagrammes de séquence sont parfois appelés diagrammes d'événements ou scénarios d'événements.



Les symboles



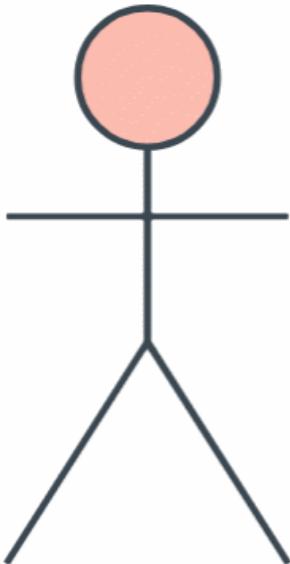
Symbole
d'objet

Représente une classe ou un objet en langage UML. Le symbole objet montre comment un objet va se comporter dans le contexte du système. Les attributs de classe ne doivent pas être énumérés dans cette forme.



Boîte
d'activation

Représente le temps nécessaire pour qu'un objet accomplisse une tâche. Plus la tâche nécessite de temps, plus la boîte d'activation est longue.



Symbole d'acteur

Montre les entités qui interagissent avec le système ou qui sont extérieures à lui.



Symbole de paquetage

Utilisé dans la notation UML 2.0 pour accueillir les éléments interactifs du diagramme. Également connue sous le nom de « cadre », cette forme rectangulaire est représentée par un petit rectangle intérieur qui contient l'intitulé du diagramme.

:User

Symbol de
ligne de vie

Représente le passage du temps qui se prolonge vers le bas. Cette ligne verticale en pointillés montre les événements séquentiels affectant un objet au cours du processus schématisé. Les lignes de vie peuvent commencer par une forme rectangulaire avec un intitulé ou par un symbole d'acteur.



Symbol de
boucle
optionnelle

On utilise ce symbole pour modéliser des scénarios où une situation qui ne se produira qu'à certaines conditions.

	<p>Symbol d'alternatives</p>	<p>Symbolise des choix (qui en général s'excluent mutuellement) entre deux séquences de messages ou plus. Pour représenter les alternatives, utilisez la forme rectangulaire comportant un intitulé et une ligne en pointillés à l'intérieur.</p>
	<p>Symbol de messages synchrones</p>	<p>Représentés par une ligne pleine terminée par une pointe de flèche pleine. On utilise ce symbole lorsqu'un expéditeur doit attendre une réponse à un message avant de continuer. Le diagramme doit montrer à la fois l'appel et la réponse.</p>



Symbole de messages asynchrones

Représentés par une ligne pleine terminée par une pointe de flèche. Les messages asynchrones ne nécessitent pas de réponse avant que l'expéditeur ne continue. Seul l'appel doit être inclus dans le diagramme.



Symbole de messages de retour asynchrones

Représentés par une ligne en pointillés terminée par une tête de flèche.



Symbole de messages de création asynchrones

Représentés par une ligne en pointillés terminée par une pointe de flèche. Ces messages créent de nouveaux objets.



Symbole de messages de réponse

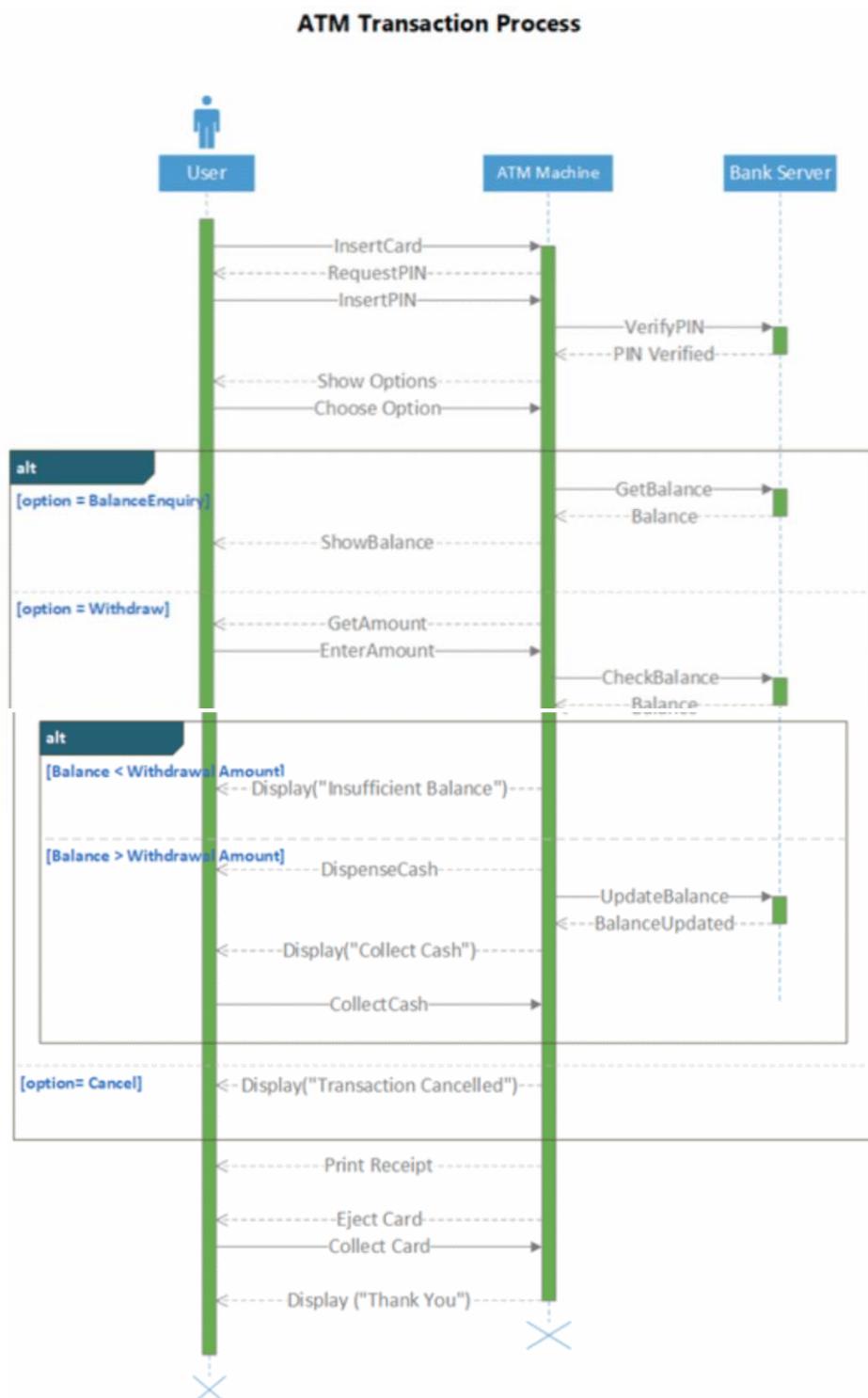
Représentés par une ligne en pointillés terminée par une pointe de flèche, ces messages sont des réponses aux appels.



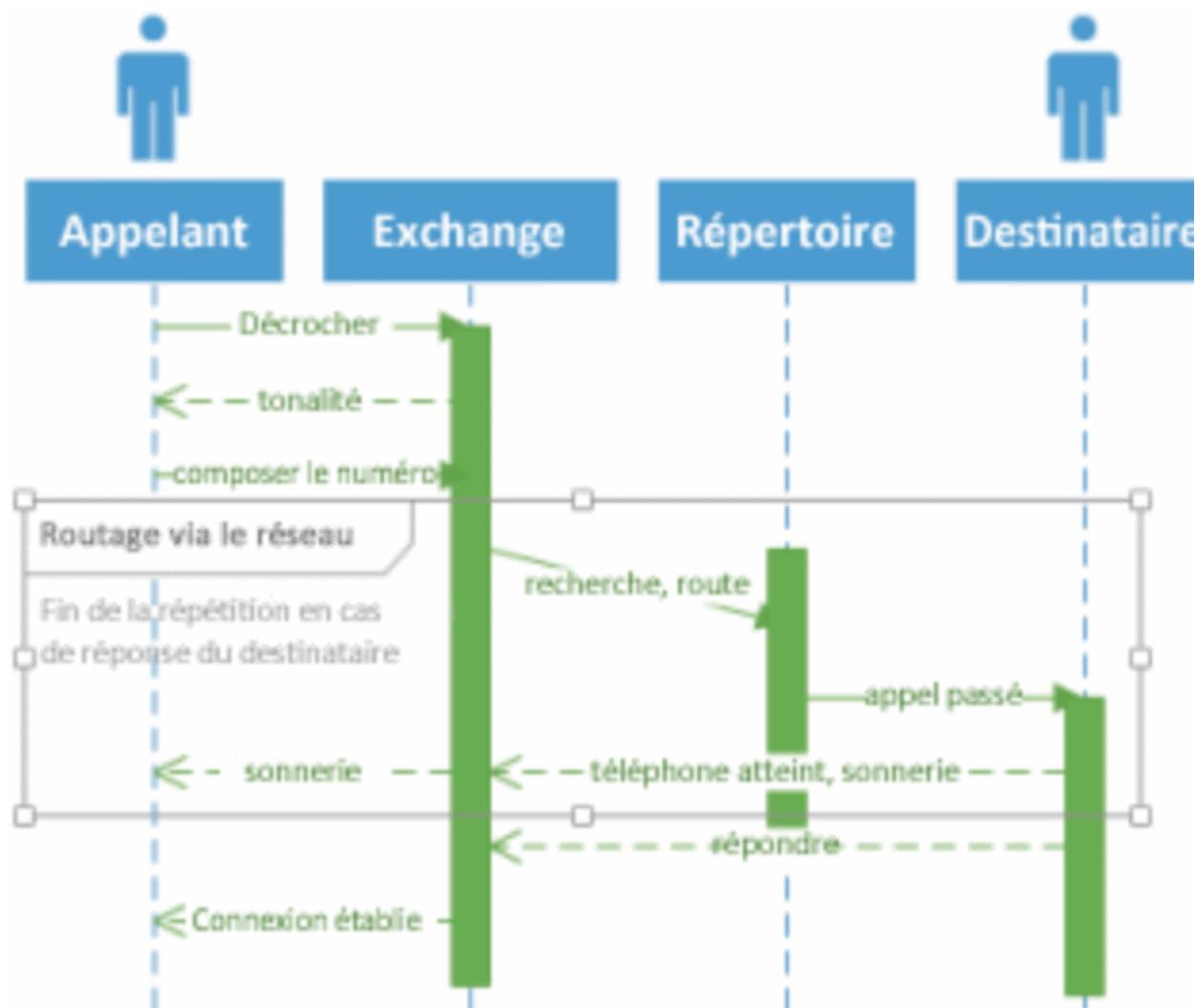
Symbole de messages de suppression

Représentés par une ligne pleine terminée par une pointe de flèche pleine, suivie du symbole X. Ces messages détruisent un objet.

Exemple: un distributeur bancaire



Exemple: un appel téléphonique:



D. Les diagrammes d'activité

Illustrent le flux de travail ou les processus dans un système. Ils sont considérés comme des diagrammes comportementaux, car ils décrivent ce qui doit arriver dans le système modélisé.

Composants de base

Action :

étape dans l'activité où les utilisateurs ou le logiciel exécutent une tâche donnée.

Nœud de décision :

embranchement conditionnel dans le flux, qui est représenté par un losange. Il comporte une seule entrée et au moins deux sorties.

Flux de contrôle :

autre nom donné aux connecteurs qui illustrent le flux entre les étapes du diagramme.

Nœud de départ :

élément symbolisant le début de l'activité, que l'on représente par un cercle noir.

Nœud de fin : élément symbolisant l'étape finale de l'activité, que l'on représente par un cercle noir avec un contour.

Les symboles:



Symbol de début

Représente le début d'un processus ou d'un flux de travail dans un diagramme d'activités. Il peut être utilisé seul ou avec un symbole de note qui explique le point de départ.

Activity

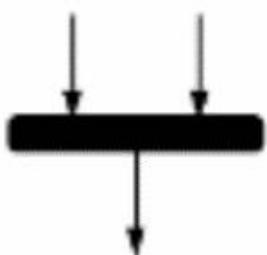
Symbole d'activité

Indique les activités qui composent un processus modélisé. Ces symboles, qui comprennent de brèves descriptions dans la forme, sont les principales composantes d'un diagramme d'activités.



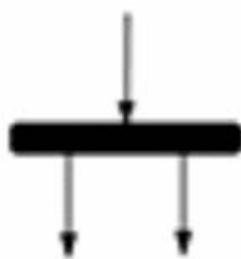
Symbole de raccord

Indique le flux directionnel, ou flux de contrôle, de l'activité. Une flèche entrante marque le début d'une étape d'une activité ; une fois l'étape terminée, le flux se poursuit avec la flèche sortante.



Symbole de raccord/barre de synchronisation

Associe deux activités simultanées et les réintroduit dans un flux où n'a lieu qu'une seule activité à la fois. Représenté par une ligne verticale ou horizontale épaisse.



Symbole d'embranchement

Divise un flux d'activités en deux activités simultanées. Symbolisé par plusieurs lignes fléchées qui partent d'un raccord.



Symbole de décision

Représente une décision et possède toujours au moins deux embranchements avec le texte de la condition pour permettre aux utilisateurs de voir les options. Ce symbole représente la ramifications ou la fusion de différents flux et sert de cadre ou de conteneur.



Remarque

Permet aux créateurs d'un diagramme ou à leurs collaborateurs de communiquer des messages supplémentaires qui n'entrent pas dans le diagramme à proprement parler. Permet de laisser des notes pour plus de clarté et de précision.



Symbole de signal d'émission

Indique qu'un signal est envoyé à une activité réceptrice.



Symbole du signal de réception

Représente l'acceptation d'un événement. Une fois l'événement reçu, le flux qui vient de cette action est terminé.



Symbol de pseudo-état historique simple

Représente une transition qui appelle le dernier état actif.



Symbol de boucle optionnelle

Permet au créateur de modéliser une séquence répétitive.



Symbole de fin de flux

Représente la fin d'un schéma de procédé spécifique. Ce symbole ne doit pas représenter la fin de tous les flux dans une activité ; dans ce cas, vous devez utiliser le symbole de fin. Le symbole de fin de flux doit être placé à la fin d'un procédé dans un flux d'activités unique.

[Condition]

Texte de condition

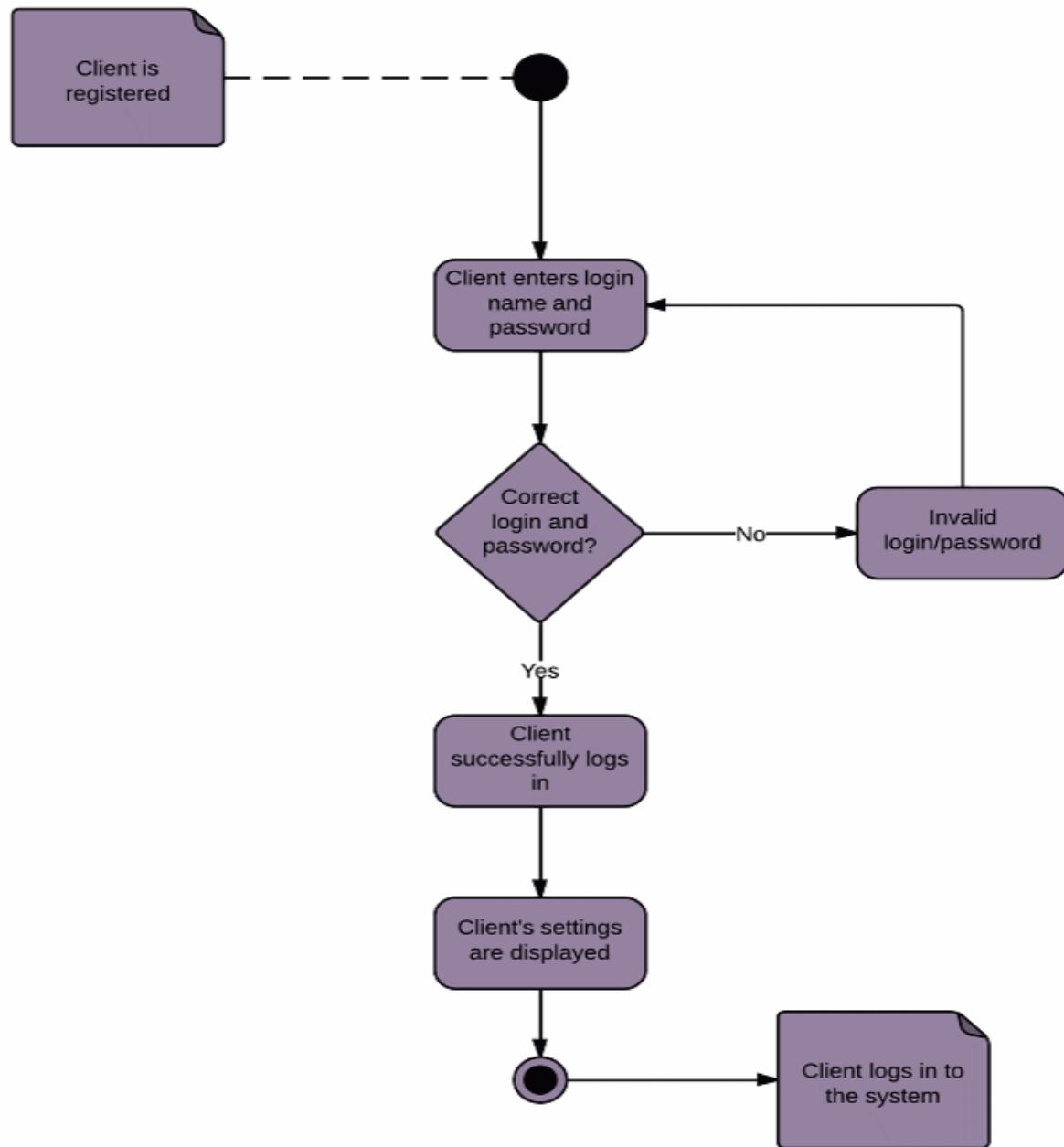
Placé à côté d'un marqueur de décision pour vous permettre de savoir dans quelle condition un flux d'activités doit se diviser dans cette direction.



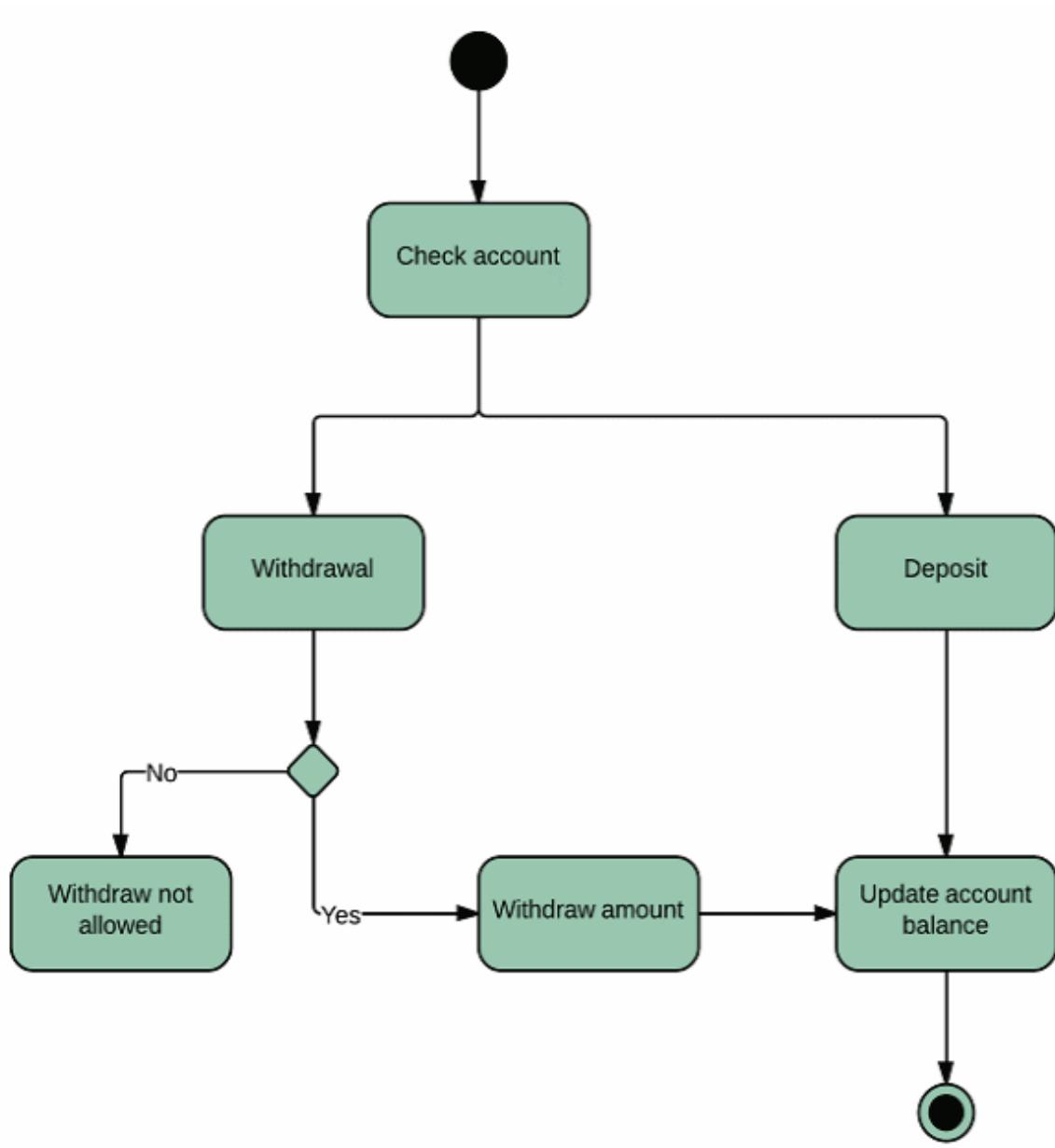
Symbole de fin

Marque l'état final d'une activité et représente l'achèvement de tous les flux d'un procédé.

Exemple: une page de connexion



Exemple2: une systeme bancaire



II. CLASSES

Une classe est un plan qui définit les propriétés (attributs) et les comportements (méthodes) d'un objet. En programmation orientée objet :

Attributs : Ce sont les variables définies dans la classe qui caractérisent les objets.

Méthodes : Ce sont les fonctions qui décrivent les comportements des objets créés à partir de la classe.

TP :

Créer un nouvel environnement python équipé de nodemon.

Dans le dossier « Lib » créer un dossier « utils » et dedans un fichier nomme « utils.py »

Dedans coller le code suivant. C'est le code d'une fonction qui permet d'afficher le nom, les propriétés et les méthodes d'un objet.

```
-----  
from pprint import pprint
```

```
def show(variable: any, name: str='') -> None:  
    """
```

*Petite fonction d'affichage
qui permet de regarder à l'intérieur des
objets ou des variables dans la console.*

Args:

*variable: La variable ou l'objet à afficher.
name (str, optional): Le nom de la variable pour l'affichage.*

Par défaut, vide.

```
    """
```

```
if hasattr(variable, '__dict__'):  
    # C'est un objet  
    if name:  
        print(f"Objet observé: {name}")  
    class_name = type(variable).__name__  
    print(f"Objet de classe: {class_name}")
```

#Propriétés de classe

```

class_attributes = {
    key: value
    for key, value in type(variable).__dict__.items()
    if not callable(value) and not key.startswith("__")
}

for elem in class_attributes:
    if isinstance(class_attributes[elem], classmethod):
        class_attributes[elem] = '<Méthode de classe>'
    if isinstance(class_attributes[elem], property):
        class_attributes[elem] = '<property object>'

if(len(class_attributes) > 0):
    print("\nPropriétés de classe:")
    pprint(class_attributes)

# Propriétés d'instance
print("\nPropriétés d'instance:")
properties = vars(variable)
pprint(properties)

# Méthodes
print("\nMéthodes:")
methods = [method for method in dir(variable)
           if callable(getattr(variable, method)) and not
method.startswith("__")]
pprint(methods)

else:
    # Ce n'est pas un objet
    if name:
        print(f"Variable observée: {name}")
    print("\nvaleur: ", end="")
    pprint(variable)

```

Concepts Clés :

- **Objet** : Une instance d'une classe. Chaque objet possède ses propres valeurs pour les attributs définis par la classe.

- **Attribut** : Une variable qui appartient à une classe ou à une instance.
- **Méthode** : Une fonction définie au sein d'une classe qui décrit le comportement des objets de cette classe.

Définir une Classe

Pour définir une classe en Python, on utilise le mot-clé **class** suivi du nom de la classe et de deux points.

Par convention comme c'est le cas pour la plupart des langages, les noms de classes en Python commencent par une majuscule. Pour respecter la nomenclature globale, utilisez le **PascalCase** pour le nom des classes.

```
main.py > ...
1   from Lib.utils import utils
2
3   class Voiture:
4       pass
5
6
7   if __name__ == "__main__":
8       pass
9
```

Instancier des Objets

Instancier signifie créer un **objet** à partir d'une classe.

L'objet est une instance concrète de la classe avec ses propres valeurs pour les attributs.

```
main.py > ...
1   from Lib.utils import utils
2
3   class Voiture:
4       pass
5
6
7   if __name__ == "__main__":
8       ma_voiture = Voiture()
9       utils.show(ma_voiture, "ma_voiture")
10
```

Sortie :

```
Objet observé: ma_voiture
Objet de classe: Voiture

Propriétés:
{}

Méthodes:
[]
```

Les Attributs

Les classes Python proposent deux types d'attributs :

- Les attributs de classe
- Les attributs d'instance

Attributs d'Instance

Les attributs d'instance sont des variables qui appartiennent à chaque instance spécifique de la classe. Chaque objet peut avoir des valeurs différentes pour ces attributs. On peut utiliser une fonction d'initialisation pour ces attributs. Les valeurs d'initialisation seront transmises à l'instanciation de la classe.

```
class Voiture:

    def __init__(self, marque, modèle): # Fonction d'initialisation (constructeur)
        self.marque = marque # Attribut d'instance
        self.modèle = modèle # Attribut d'instance
```

```
if __name__ == "__main__":
    ma_voiture = Voiture("Toyota", "Corolla") # instanciation
    utils.show(ma_voiture, "ma_voiture") # debug
```

Sortie :

```
Objet observé: ma_voiture
Objet de classe: Voiture

Propriétés:
{'marque': 'Toyota', 'modele': 'Corolla'}

Méthodes:
[]
```

Dans la fonction d'initialisation,

```
def __init__(self, marque, modèle): # Fonction d'initialisation (constructeur)
```

`__init__` est une méthode spéciale appelée **constructeur** qui initialise les attributs de l'objet.

Le paramètre « **self** » est une référence à l'instance actuelle de la classe (i.e. l'objet créé à partir de la classe). Self est toujours le 1^{er} paramètre.

Lors de l'instanciation on ne fournit les valeurs que pour les autres paramètres dans l'ordre. Le paramètre « **self** » est géré automatiquement par Python lui-même.

Attributs de Classe

Les attributs de classe sont des variables partagées par toutes les instances de la classe. Ils sont **déclarés ET initialisés** directement dans la classe, en dehors de toute méthode.

```
ain.py > ...
from Lib.utils import utils

class Voiture:
    roues = 4 # Attribut de classe

    def __init__(self, marque, modèle): # Fonction d'initialisation (constructeur)
        self.marque = marque # Attribut d'instance
        self.modèle = modèle # Attribut d'instance

    if __name__ == "__main__":
        ma_voiture = Voiture("Toyota", "Corolla")
        utils.show(ma_voiture, "ma_voiture")
```

Sortie :

```
Objet observé: ma_voiture
Objet de classe: Voiture

Propriétés de classe:
{'roues': 4}

Propriétés d'instance:
{'marque': 'Toyota', 'modèle': 'Corolla'}

Méthodes:
[]
```

roues est un attribut de classe. Toutes les instances de Voiture auront 4 roues par défaut.

Particularité des propriétés de classe : On peut y accéder via la classe elle-même. C'est la version Python d'un attribut statique. Dans python un membre statique est accessible via l'objet **ET** la classe...

```
if __name__ == "__main__":
    ma_voiture = Voiture("Toyota", "Corolla") # instantiation
    utils.show(ma_voiture, "ma_voiture") # debug

    # acces à une propriété de classe
    print("\nAcces à une propriété de classe:")
    print("Voiture.roues: ", Voiture.roues) # sur la classe elle-même
    print("ma_voiture.roues: ", ma_voiture.roues) # sur une instance de la classe
```

```
Acces à une propriété de classe:
Voiture.roues:  4
ma_voiture.roues:  4
```

En revanche on ne peut accéder à des propriétés d'instance que sur un objet issu de l'instanciation de la classe :

```
# acces à une propriété d'instance
print("\nAcces à une propriété d'instance:")
print(f"Marque de ma voiture (ma_voiture.marque): {ma_voiture.marque}")
```

Sortie :

```
Acces à une propriété d'instance:
Marque de ma voiture (ma_voiture.marque): Toyota
```

Les Méthodes

Les méthodes sont des fonctions définies au sein d'une classe qui décrivent les comportements des objets de cette classe.

On distingue également :

- Les méthodes d'instances
- Les méthodes de classe

Méthodes d'Instance

Les méthodes d'instance opèrent sur une instance spécifique de la classe (sur l'objet créé à partir de la classe). Elles prennent au minimum un paramètre : « self » ce qui leur permet d'accéder aux propriétés d'instance.

```
class Voiture:  
    roues = 4  
  
    def __init__(self, marque, modèle): # Fonction d'initialisation (constructeur)  
        self.marque = marque # Attribut d'instance  
        self.modèle = modèle # Attribut d'instance  
  
    def afficher_details(self): #methode d'instance  
        print(f"Ma voiture est une {self.marque}, Modèle: {self.modèle}")
```

```
# appel d'une methode d'instance  
print("\nAppel d'une methode d'instance:")  
ma_voiture.afficher_details()
```

Sortie :

```
Appel d'une methode d'instance:  
Ma voiture est une Toyota, Modèle: Corolla
```

Méthodes de Classe

Les méthodes de classe opèrent sur la classe elle-même plutôt que sur une instance. Elles sont définies avec le décorateur `@classmethod` et reçoivent la classe comme premier argument (`cls`) qui permet de référencer les propriétés de classe.

Elles sont appelables sur la classe ou sur l'objet issue de cette classe.

```
class Voiture:
    roues = 4

    def __init__(self, marque, modèle): # Fonction d'initialisation (constructeur)
        self.marque = marque # Attribut d'instance
        self.modèle = modèle # Attribut d'instance

    def afficher_details(self): #méthode d'instance
        print(f"Ma voiture est une {self.marque}, Modèle: {self.modèle}")

    @classmethod
    def afficher_nombre_de_roues(cls):
        print(f"Elle a {cls.roues} roues")
```

```
# appel d'une méthode de classe
print("\nAppel d'une méthode de classe sur la classe puis sur l'objet:")
Voiture.afficher_nombre_de_roues() # sur la classe elle-même
ma_voiture.afficher_nombre_de_roues() # sur une instance de la classe
```

Sortie :

```
Appel d'une méthode de classe sur la classe puis sur l'objet:
Elle a 4 roues
Elle a 4 roues
```

Les Méthodes spéciales

Les **méthodes spéciales** en Python sont des méthodes prédéfinies qui commencent et se terminent par des doubles underscores (`__`). Elles permettent de définir des comportements spécifiques pour les objets.

Le constructeur `__init__`

Le constructeur est une méthode spéciale appelée lors de la création d'une instance de la classe. Il est généralement utilisé pour initialiser les attributs de l'objet. C'est à la base une fonction.

Le constructeur obéit donc aux règles qui régissent toutes les fonctions en python – notamment celles qui concernent les paramètres. Rajoutons un paramètre optionnel dans notre constructeur :

```
class Voiture:  
    roues = 4  
  
    def __init__(self, marque, modèle, vitesse=0): # Fonction d'initialisation (constructeur)  
        self.marque = marque # Attribut d'instance  
        self.modèle = modèle # Attribut d'instance  
        self.vitesse = vitesse # Attribut d'instance avec valeur par défaut
```

```
if __name__ == "__main__":  
    ma_voiture = Voiture("Toyota", "Corolla") # instantiation  
    utils.show(ma_voiture, "ma_voiture") # debug
```

Sortie :

```
Objet observé: ma_voiture  
Objet de classe: Voiture  
  
Propriétés de classe:  
{'afficher_nombre_de_roues': '<Méthode de classe>', 'roues': 4}  
  
Propriétés d'instance:  
{'marque': 'Toyota', 'modèle': 'Corolla', 'vitesse': 0}  
  
Méthodes:  
['afficher_details', 'afficher_nombre_de_roues']
```

La méthode `__new__`

Cette méthode est très rarement utilisée. Elle est automatiquement exécutée **AVANT** `__init__` lors de l'instanciation d'une classe.

Pour visualiser ces appels créer un dossier « classes » dans le dossier « Lib » et dedans créer un fichier nommé « maclasse.py »

Dans ce fichier écrire le code suivant :

```
classes > {} maclasse.py > ...
class MaClasse:
    def __new__(cls, *args, **kwargs):
        print("Appel de __new__")
        instance = super(MaClasse, cls).__new__(cls)
        return instance

    def __init__(self, *args, **kwargs):
        print("Appel de __init__")
        self.props = args[0]
```

Puis dans `main.py` importer `MaClasse` :

```
main.py > ...
1  from Lib.utils import utils
2  from Lib.classes.maclasse import MaClasse
3
```

Puis instancions `MaClasse` :

```
# L'appel de __new__ intervient juste avant celui de __init__
print("\nMéthodes spéciales appelées lors de l'instanciation: ")
ma_classe = MaClasse("prop1")
```

Sortie :

```
Méthodes spéciales appelées lors de l'instanciation:
Appel de __new__
Appel de __init__
```

La méthode `__str__`

Permet de définir la représentation en chaîne de caractères (**stringifier**) de l'objet pour l'affichage via `print()`.

```
def __str__(self):
    return f"la {self.marque} {self.modele} avance à {self.vitesse} km/h"
```

```
# Appel à la méthode spéciale __str__ de l'objet:
print("\nAppel à la méthode spéciale __str__ de l'objet:")
print(ma_voiture)
```

```
Appel à la méthode spéciale __str__ de l'objet:
la Toyota Corolla avance à 0 km/h
```

La méthode `__repr__`

Définir la représentation officielle de l'objet, utilisée par `repr()`.

```
def __repr__(self):
    return f"Voiture(marque='{self.marque}', modele='{self.modele}', vitesse={self.vitesse})"
```

```
# Appel à la méthode spéciale __repr__ de l'objet:
print("\nAppel à la méthode spéciale __repr__ de l'objet:")
print(repr(ma_voiture))
```

```
Appel à la méthode spéciale __repr__ de l'objet:
Voiture(marque='Toyota', modele='Corolla', vitesse=0)
```

La méthode `__del__`

Finaliseur de l'instance. Cette méthode est appelée lorsqu'un objet est sur le point d'être détruit, mais n'est pas responsable de sa destruction.

```
def __del__(self):
    print('\nsuppression de l\'objet... bye!')
```

Sortie :

A la fin du script l'objet est détruit. Juste avant la destruction, la méthode `__del__(self)` est appelée.

```
suppression de l'objet... bye!
[nodemon] clean exit - waiting for changes before restart
```

Encapsulation

L'encapsulation est un principe de la POO qui consiste à restreindre l'accès direct à certains composants d'un objet et à fournir des méthodes pour interagir avec ces composants. En Python, cela est généralement réalisé en utilisant des attributs privés (prefixés par un underscore `_` ou double underscore `__`).

Attributs Privés

- **Attributs avec un seul underscore (`_`)** : Indiquent que l'attribut est destiné à être utilisé en interne et ne devrait pas être accédé directement.
- **Attributs avec double underscore (`__`)** : En Python, cela déclenche la **name mangling**, rendant l'attribut moins accessible depuis l'extérieur.

Dans notre code rendons l'attribut « vitesse » privé :

```
self.__vitesse = vitesse # Attribut d'instance PRIVE avec valeur par défaut
```

Puis rajoutons quelques méthodes :

```
def accelerer(self, increment):
    self.__vitesse += increment

def freiner(self, decrement):
    if self.__vitesse > decrement:
        self.__vitesse -= decrement
    else:
        print("La vitesse ne peut pas être négative.")

def get_vitesse(self):
    return self.__vitesse

def set_vitesse(self, vitesse):
    if vitesse >= 0:
        self.__vitesse = vitesse
    else:
        print("La vitesse ne peut pas être négative.")
```

Et modifions les méthodes utilisant précédemment la propriété « vitesse » pour ne pas lever d'erreur :

```
def __str__(self):
    return f"la {self.marque} {self.modele} avance à {self.__vitesse} km/h"

def __repr__(self):
    return f"Voiture(marque='{self.marque}', modele='{self.modele}', vitesse={self.__vitesse})"
```

Puis manipulons notre voiture :

```
# Maniement de l'objet ma_voiture
print("\nManiement de l'objet ma_voiture:")
print("affichage de la vitesse initiale - ma_voiture.get_vitesse():")
print(ma_voiture.get_vitesse(), 'km/h')

print("\nacceleration de 50km/h - ma_voiture.accelerer(50):")
ma_voiture.accelerer(50)

print("affichage de la vitesse - ma_voiture.get_vitesse():")
print(ma_voiture.get_vitesse(), 'km/h')

print("\nReglage de la vitesse à 100km/h - ma_voiture.set_vitesse(100):")
ma_voiture.set_vitesse(100)

print("affichage de la vitesse - ma_voiture.get_vitesse():")
print(ma_voiture.get_vitesse(), 'km/h')

print("\nTentative de rendre la vitesse négative - ma_voiture.set_vitesse(-10):")
ma_voiture.set_vitesse(-10)

print("\nTentative de rendre la vitesse négative - ma_voiture.freiner(101):")
ma_voiture.freiner(101)
```

Sortie :

```
Maniement de l'objet ma_voiture:
affichage de la vitesse initiale - ma_voiture.get_vitesse():
0 km/h

acceleration de 50km/h - ma_voiture.accelerer(50):
affichage de la vitesse - ma_voiture.get_vitesse():
50 km/h

Reglage de la vitesse à 100km/h - ma_voiture.set_vitesse(100):
affichage de la vitesse - ma_voiture.get_vitesse():
100 km/h

Tentative de rendre la vitesse négative - ma_voiture.set_vitesse(-10):
La vitesse ne peut pas être négative.

Tentative de rendre la vitesse négative - ma_voiture.freiner(101):
La vitesse ne peut pas être négative.

suppression de l'objet... bye!
[nodemon] clean exit - waiting for changes before restart
```

III. HERITAGE

L'**héritage** permet de créer une nouvelle classe à partir d'une classe existante. La classe dérivée (ou enfant) hérite des attributs et des méthodes de la classe de base (ou parent), tout en pouvant ajouter de nouvelles fonctionnalités ou modifier celles existantes.

Une voiture est un véhicule. Si nous créons une classe « Vehicule » renfermant toutes les props et méthodes que tous les véhicules possèdent, alors nous pouvons faire notre classe « Voiture » hériter de « Vehicule »

Créer un nouvel environnement Python3

Dans le dossier Lib coller le dossier « utils » créé dans le chapitre précédent. Créer dans Lib un dossier « classes »

Dans class créer deux fichiers :

- vehicule.py
- voiture.py

Dans vehicule.py définissons une classe « Vehicule » :

```
class Vehicule:  
    def __init__(self, marque, modele):  
        self.marque = marque  
        self.modele = modele  
  
    def afficher_details(self):  
        print(f"Marque: {self.marque}, Modèle: {self.modele}")
```

Puis dans voiture.py définissons une classe « Voiture » qui va hériter de « Vehicule » :

```
from Lib.classes.vehicule import Vehicule

class Voiture(Vehicule):
    def __init__(self, marque, modele, vitesse=0):
        super().__init__(marque, modele)
        self.vitesse = vitesse

    def accelerer(self, increment):
        self.vitesse += increment

    def freiner(self, decrement):
        self.vitesse -= decrement

    def afficher_details(self):
        super().afficher_details()
        print(f"Vitesse: {self.vitesse} km/h")
```

```
class Voiture(Vehicule):
```

Voiture hérite de Vehicule en plaçant le nom de la classe parent entre parenthèses.

```
super().__init__(marque, modele)
```

super().__init__(marque, modele) appelle le constructeur de la classe parent pour initialiser les attributs hérités.

```
def afficher_details(self):
    super().afficher_details()
    print(f"Vitesse: {self.vitesse} km/h")
```

La méthode `afficher_details` est redéfinie ou encore surchargée (`overridden`) pour inclure l'attribut `vitesse` en plus de la marque et du modèle. En terme de surcharge python permet de redefinir entièrement la méthode du parent ou bien d'inclure un appel a la méthode du parent pour lui rajouter des instructions comme dans l'exemple ci-dessus.

Particularité de Python : L'héritage multiple

Python permet a une classe d'hériter de multiples classes parente. Ce mécanisme est impossible dans de nombreux autres langages majeurs (JS, PHP, JAVA, etc...).

Créons une classe « Moteur » dans son fichier propre (Lib/classes/moteur.py)

```
class Moteur:
    def demarrer_moteur(self):
        print("Le moteur démarre.")
```

Puis importons cette classe dans le fichier de la classe Voiture afin qu'elle en hérite :

```
from Lib.classes.vehicule import Vehicule
from Lib.classes.moteur import Moteur

class Voiture(Vehicule, Moteur):
```

Enfin dans le fichier main.py instancions une voiture et utilisons-la.

```
from Lib.utils import utils
from Lib.classes.voiture import Voiture

if __name__ == "__main__":
    ma_voiture = Voiture("Porshe", "Cayenne", 120)
    utils.show(ma_voiture, 'ma_voiture')

    ma_voiture.demarrer_moteur() # méthode héritée
    ma_voiture.afficher_details() # méthode héritée et surchargée
```

Sortie :

```
Objet observé: ma_voiture
Objet de classe: Voiture

Propriétés d'instance:
{'marque': 'Porshe', 'modele': 'Cayenne', 'vitesse': 120}

Méthodes:
['accelerer', 'afficher_details', 'demarrer_moteur', 'freiner']
Le moteur démarre.
Marque: Porshe, Modèle: Cayenne
Vitesse: 120 km/h
```

Le problème avec l'héritage multiple, c'est que si deux parents possèdent une méthode dont la signature est identique, il est impossible de savoir laquelle des deux méthodes sera exécutée si elle est appelée sur un objet issu de la classe fille. C'est d'ailleurs pour cela que l'héritage multiple est interdit dans la majeure parties des langages de programmation.

Voyons comment Python gère ce problème.

Supposons que la classe Moteur possède une méthode « afficher_details() » qui a la même signature de celle que renferme la classe « Véhicule. »

```
class Moteur:  
    def demarrer_moteur(self):  
        print("Le moteur démarre.")  
  
    def afficher_details(self):  
        print("Je suis un superbe moteur V8")
```

Lorsqu'on appelle « afficher_details » sur l'objet Voiture

```
class Voiture(Vehicule, Moteur):  
    def __init__(self, marque, model  
                 vitesse):  
        super().__init__(marque, model)
```

```
Le moteur démarre.  
Marque: Porshe, Modèle: Cayenne  
Vitesse: 120 km/h
```

C'est la méthode de véhicule qui s'exprime.

Si nous inversons l'ordre d'héritage :

```
class Voiture(Moteur, Vehicule):  
    def __init__(self, marque, model  
                 vitesse):  
        super().__init__(marque, model)
```

```
Le moteur démarre.  
Je suis un superbe moteur V8  
Vitesse: 120 km/h
```

C'est **l'ordre d'héritage** qui détermine la méthode qui sera appelée

La notion de polymorphisme

Le **polymorphisme** permet d'utiliser des méthodes de manière interchangeable, même si elles proviennent de classes différentes, tant qu'elles respectent le même **contrat** c'est-à-dire des signatures de méthode identiques.

Créer une classe Animal, une classe Chien et une classe Chat qui héritent de Animal.

```
class Animal:  
    def parler(self):  
        # méthode destinée à être implementée  
        # par les classes filles  
        pass
```

```
from Lib.classes.animal import Animal  
  
class Chien(Animal):  
    def parler(self): # méthode surchargée  
        print("Woof!")
```

```
from Lib.classes.animal import Animal  
  
class Chat(Animal):  
    def parler(self): # méthode surchargée  
        print("Meow!")
```

Puis dans utils.py rajoutons une nouvelle fonction qui prendra un objet de classe Animal en argument.

Les Classes sont des **TYPES de données** et un objet de classe Chien aura le type **Chien ET** le type **Animal**.

Rajoutons dans utils.py :

```
from Lib.classes.animal import Animal
from pprint import pprint

def faire_parler_animal(animal: Animal) -> None:
    """
        Méthode permettant d'appeler la méthode parler
        d'un objet de classe ou héritant de la classe
        Animal

    args:
        animal: objet de classe Animal
    """
    show(animal, 'animal')
    animal.parler()
```

Puis :

Revenons dans main.py et importons les deux classes que nous manipulerons : Chien et Chat.

```
from Lib.classes.chien import Chien
from Lib.classes.chat import Chat
```

```
#instanciation d'un chien
chien = Chien()
|
    (variable) chat: Chat
chat = Chat()

#utilisation de nos deux animaux
print("\nfaisons parler l'objet chien")
utils.faire_parler_animal(chien)

print("\nfaisons parler l'objet chat")
utils.faire_parler_animal(chat)
```

Sortie :

```
faisons parler l'objet chien
Objet observé: animal
Objet de classe: Chien

Propriétés d'instance:
{}

Méthodes:
['parler']
Woof!
```

```
faisons parler l'objet chat
Objet observé: animal
Objet de classe: Chat

Propriétés d'instance:
{}

Méthodes:
['parler']
Meow!
```

Bonnes Pratiques de conception Orientée objet

1. Nommage des Classes :

- Utilisez des noms en **CamelCase** pour les classes (ex. Voiture, Animal).

2. Encapsulation :

- Utilisez des attributs privés pour protéger les données internes des objets.
- Fournissez des **accesseurs** (getters) et **mutateurs** (setters) pour manipuler les attributs privés.

3. Documentation :

- Documentez vos classes et méthodes en utilisant des **docstrings** pour améliorer la lisibilité et la maintenabilité du code.

4. Respect des Principes SOLID :

- **Single Responsibility Principle** : Chaque classe doit avoir une seule responsabilité.
- **Open/Closed Principle** : Les classes doivent être ouvertes à l'extension mais fermées à la modification.
- **Liskov Substitution Principle** : Les objets des classes dérivées doivent pouvoir remplacer les objets des classes de base.
- **Interface Segregation Principle** : Préférez plusieurs interfaces spécifiques à une interface générale.
- **Dependency Inversion Principle** : Dépendre des abstractions plutôt que des concrétions.

5. Utilisation de super() :

- Lors de l'héritage, utilisez super() pour appeler les méthodes de la classe parente de manière propre et maintenable.

6. Eviter les Attributs Globaux :

- Limitez l'utilisation d'attributs globaux pour maintenir le code encapsulé et éviter les effets de bord.

IV. ENCAPSULATION

L'**encapsulation** est l'un des quatre principes fondamentaux de la POO, aux côtés de l'abstraction, de l'héritage et du polymorphisme. Elle consiste à **masquer les détails internes** d'une classe et à **fournir une interface contrôlée** pour interagir avec ces détails.

En d'autres termes, l'encapsulation protège les données internes d'une classe en limitant l'accès direct aux attributs et méthodes, et en offrant des méthodes publiques (souvent appelées **accesseurs** et **mutateurs**) pour manipuler ces données.

Créer un nouvel environnement Python3 et récupérer dedans le dossier Lib/utils/utils.py.

Ne garder dans utils.py que la méthode show().

Créer une nouvelle classe nommée Access

```
class Access:
    def __init__(self):
        self.__attribut_private = "Accessible uniquement dans la classe"

        self.__attribut_protected = "Accessible dans la classe et ses
        sous-classes"

        self.__attribut_private = "Accessible uniquement dans la classe"

    def methode_public(self):
        print("Méthode publique")

    def _methode_protected(self):
        print("Méthode protégée")

    def __methode_private(self):
        print("Méthode privée")

    def afficher_private(self):
        print(self.__attribut_private)
        self.__methode_private()
```

Niveaux d'Accès en Python

Contrairement à d'autres langages orientés objet comme Java ou C++, Python **ne possède pas de modificateurs d'accès stricts** (comme **public**, **protected**, **private**).

Cependant, Python utilise des **conventions de nommage** pour indiquer l'intention de l'accès aux attributs et méthodes.

Public

Les attributs et méthodes publics sont accessibles depuis n'importe où, que ce soit à l'intérieur ou à l'extérieur de la classe.

- **Convention** : Noms sans préfixe particulier.

```
def __init__(self):
    self.__attribut_private = "Accessible uniquement dans la classe"

def __methode_private(self):
    print("Méthode privée")
```

Protected

Les attributs et méthodes protégés sont destinés à être accessibles uniquement à l'intérieur de la classe et de ses sous-classes.

- **Convention** : Noms précédés d'un seul underscore _.

```
self.__attribut_protected = "Accessible dans la classe et ses
sous-classes"
```

```
def __methode_protected(self):
    print("Méthode protégée")
```

Private

Les attributs et méthodes privés sont destinés à être accessibles uniquement à l'intérieur de la classe où ils sont définis.

- **Convention** : Noms précédés de deux underscores `__`.

```
self.__attribut_private = "Accessible uniquement dans la classe"
```

```
def __methode_private(self):
    print("Méthode privée")
```

Attention !! :

En Python, comme en Javascript, les attributs privés sont **pseudo-privés**. Python applique une technique appelée **name mangling** pour rendre l'accès extérieur plus difficile, mais ce n'est pas une véritable encapsulation.

Le **name mangling** est un mécanisme de Python qui modifie le nom des attributs privés pour les rendre moins accessibles de l'extérieur de la classe. Lorsque vous préfixez un attribut ou une méthode avec deux underscores `__`, Python renomme automatiquement cet attribut en `_NomDeLaClasse__NomAttribut`.

Allons dans `main.py` et importons la classe Access :

```
from Lib.classes.access import Access
```

Puis manipulons pour voir si nous pouvons avoir accès aux différentes propriétés et méthodes.

```
# Accès sur l'objet à une propriété protected
print("\nAccès direct sur l'objet à une propriété protected:")
print("possible... mais vraiment pas recommandé...")
print(access._attribut_protected)
```

```
Acces direct sur l'objet a une propriété protected:  
possible... mais vraiment pas recommandé...  
Accessible dans la classe et ses sous-classes
```

On a un accès direct sur l'objet aux méthodes protected en python, comme en JS.

```
# Acces sur l'objet a une méthode protected  
print("\nAcces direct sur l'objet a une méthode protected:")  
print("possible... mais vraiment pas recommandé...")  
access._methode_protected()
```

```
Acces direct sur l'objet a une méthode protected:  
possible... mais vraiment pas recommandé...  
Méthode protégée
```

Pareil pour les méthodes protected.

Voyons pour un accès direct sur l'objet à une propriété private :

```
# Acces sur l'objet a une propriété private  
print("\nAcces direct sur l'objet a une propriété private:")  
print("Grace au Name Mangling, cela leve une exception:")  
print(access._attribut_private)
```

```
Acces direct sur l'objet a une propriété protected:  
Grace au Name Mangling, cela leve une exception:  
Traceback (most recent call last):  
  File "G:\Desktop\tutos\python\vscode\pythonpoo\03-encapsulation\m  
6, in <module>  
    print(access._attribut_private)  
    ^^^^^^^^^^^^^^^^^^  
AttributeError: 'Access' object has no attribute '_attribut_private'  
: '_attribut_protected'?  
[nodemon] app crashed - waiting for file changes before starting...
```

Commenter la ligne

```
print("Grace au Name Mangling, cela leve une exception")
# print(access._attribut_private)
```

Testons sur une méthode private :

```
# Acces sur l'objet à une méthode private
print("\nAcces direct sur l'objet à une méthode private:")
print("Grace au Name Mangling, cela leve une exception:")
access._methode_private()
```

```
Acces direct sur l'objet à une méthode private:
Grace au Name Mangling, cela leve une exception:
Traceback (most recent call last):
  File "G:\Desktop\tutos\python\vscode\pythonpoo\03-encapsulation\m1.py", in <module>
    access._methode_private()
    ^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Access' object has no attribute '_methode_private'
 '_methode_protected'?
[nodemon] app crashed - waiting for file changes before starting...
```

Accès normal à un membre privé à travers une méthode getter publique :

```
# Acces à un membre privé à travers un getter
print("\nAcces à un membre privé à travers un getter:")
print("Méthode correcte d'accès à un membre privé...")
access.afficher_private()
```

```
Acces à un membre privé à travers un getter:
Méthode correcte d'accès à un membre privé...
Accessible uniquement dans la classe
Méthode privée
```

Détournement du Name Mangling pour qd même accéder à un membre privé directement sur l'objet :

```
# détournement du Name Mangling pour acceder de manière directe
# sur l'objet a un membre prive en utilisant la convention de
# nommage du name mangling object._classname__attribut_private
# et object._classname_methode_private()
print("\ndétournement du Name Mangling pour acceder de manière
directe")
print("sur l'objet a un membre prive en utilisant la convention de ")
print("nommage du name mangling object._classname__attribut_private")
print("et object._classname_methode_private()")
print(access._Access__attribut_private)
access._Access__methode_private()
```

détournement du Name Mangling pour acceder de manière directe
sur l'objet a un membre prive en utilisant la convention de
nommage du name mangling object._classname__attribut_private
et object._classname_methode_private()
Accessible uniquement dans la classe
Méthode privée

Notez bien :

Ne pas utiliser le name mangling comme une fonctionnalité régulière.
C'est une protection minimale destinée à éviter les accès
accidentels, mais elle ne doit pas être considérée comme une
véritable sécurité. **Utilisez des GETTERS et SETTERS**

Préférence pour les conventions : Utilisez les conventions _ et __
pour indiquer l'intention d'accès, mais ne vous fiez pas uniquement à
elles pour la sécurité.

Getters et Setters

Pour une meilleure encapsulation et un contrôle accru sur l'accès et la modification des attributs, Python propose les **propriétés** à l'aide du décorateur `@property`. Cela permet de définir des accesseurs (getter) et des mutateurs (setter) tout en accédant aux attributs comme s'ils étaient publics.

Créer une classe « **Voiture** » dans son fichier propre.

```
class Voiture:
    def __init__(self, marque, modèle):
        self.__marque = marque
        self.__modèle = modèle

    # Getter pour marque
    @property
    def marque(self):
        return self.__marque

    # Setter pour marque
    @marque.setter
    def marque(self, valeur):
        if valeur:
            self.__marque = valeur
        else:
            raise ValueError("La marque ne peut pas être vide.")

    # Getter pour modèle
    @property
    def modèle(self):
        return self.__modèle

    # Setter pour modèle
    @modèle.setter
    def modèle(self, valeur):
        if valeur:
            self.__modèle = valeur
        else:
            raise ValueError("Le modèle ne peut pas être vide.")
```

Le mécanisme des Propriétés permet :

- **Une Encapsulation Renforcée** : Contrôle strict sur la façon dont les attributs sont accédés et modifiés.
- **La Validation des Données** : Possibilité de valider les valeurs avant de les assigner.
- **Plus de Flexibilité** : Permet de changer l'implémentation interne sans affecter le code extérieur.
- **Une Interface Consistante** : L'accès aux attributs reste similaire à celui des attributs publics.

Testons notre voiture :

- Accès aux membres privé par getters

```
from Lib.classes.voiture import Voiture
```

```
# on instancie Voiture
voiture1 = Voiture("Toyota", "Corolla")
utils.show(voiture1, "voiture1")

# Accès via les getters
print("\nAccès via les getters setters:")
print(voiture1.marque)
print(voiture1.modele)
```

Sortie :

```
Objet observé: voiture1
Objet de classe: Voiture

Propriétés de classe:
{'marque': '<property object>', 'modele': '<property object>'}

Propriétés d'instance:
{'_Voiture__marque': 'Toyota', '_Voiture__modele': 'Corolla'}

Méthodes:
[]

Accès via les getters setters:
Toyota
Corolla
```

Le décorateur @Property a créé deux objets de type Property à partir des deux attributs privés.

- Modifications de membres privés par setters

```
# Modification via les setters
print("\nModification via les setters:")
voiture1.marque = "Honda"
voiture1.modele = "Civic"

print(voiture1.marque)
print(voiture1.modele)
```

```
Modification via les setters:
```

```
Honda
```

```
Civic
```

Tentative de modification avec une valeur invalide

```
# Tentative de modification avec une valeur invalide
print("\nTentative de modification avec une valeur invalide:")
print("Leve une exception ValueError...")
voiture1.marque = ""|
```

```
Tentative de modification avec une valeur invalide:
```

```
Leve une exception ValueError...
```

```
Traceback (most recent call last):
```

```
  File "G:\Desktop\tutos\python\vscode\pythonpoo\03-encapsulation.py", in <module>
    voiture1.marque = ""|
          ^^^^^^^^^^^^^^
```

```
    File "G:\Desktop\tutos\python\vscode\pythonpoo\03-encapsulation.py", line 17, in marque
        raise ValueError("La marque ne peut pas être vide.")
```

```
ValueError: La marque ne peut pas être vide.
```

```
[nodemon] app crashed - waiting for file changes before start
```

Bonnes Pratiques d'Encapsulation en Python

1. Utiliser des Attributs Privés pour les Données Internes :

- Préférez utiliser des attributs privés (`__attribut`) pour les données sensibles ou critiques.

2. Fournir des Méthodes d'Accès :

- Utilisez des méthodes `@property` pour contrôler l'accès et la modification des attributs.

3. Respecter les Conventions de Nommage :

- Utilisez un seul underscore _ pour les attributs et méthodes protégés.
- Utilisez deux underscores __ pour les attributs et méthodes privés.

4. Limiter l'Accès Direct aux Attributs :

- Évitez d'accéder ou de modifier directement les attributs privés depuis l'extérieur de la classe.

5. Documenter les Attributs et Méthodes :

- Utilisez des docstrings pour expliquer l'intention et l'utilisation des attributs et méthodes.

6. Utiliser des Méthodes Spécifiques pour les Opérations Complexes :

- Plutôt que d'exposer les attributs internes, fournissez des méthodes pour effectuer des opérations complexes ou sensibles.

7. Éviter le Name Mangling comme Solution de Sécurité :

- Ne comptez pas sur le name mangling pour protéger les données sensibles. C'est une mesure de dissuasion minimale.

8. Séparer les Responsabilités :

- Chaque classe devrait avoir une responsabilité claire, minimisant la nécessité d'accéder directement aux attributs internes.

Exercices avec corrigés

1- Créer une classe « Personne »

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self._age = age
        self.__numero_securite = "123-45-6789"

    def afficher_info(self):
        print(f"Nom: {self.nom}, Âge: {self._age}, Numéro de sécurité: {self.__numero_securite}")
```

- Créer une instance de Personne
- Accéder aux attributs de manière directe sur l'objet créé
- Accéder directement aux attributs privé par Name Mangling
- Modifier de manière directe tous les attributs depuis l'objet
- Appeler la méthode de l'objet

Correction :

```
# -----
# Exercice 1 - classe Personne
# -----


# Création d'une instance
personne = Personne("Alice", 30)

# Accès aux attributs
print(personne.nom)
print(personne._age)
# print(personne.__numero_securite) # Provoque une AttributeError

# Accès via name mangling
print(personne._Personne__numero_securite) # Output: 123-45-6789

# Modification des attributs
personne.nom = "Bob"          # Valide
personne._age = 31             # Valide, mais déconseillé
personne._Personne__numero_securite = "987-65-4321" # Possible mais
déconseillé

# Affichage des informations
personne.afficher_info()
```

Sortie :

```
Alice  
30  
123-45-6789  
Nom: Bob, Âge: 31, Numéro de sécurité: 987-65-4321
```

2- Créer une classe « CompteBancaire »

```
class CompteBancaire:  
    def __init__(self, titulaire, solde=0):  
        self.titulaire = titulaire  
        self._solde = solde  
        self.__transactions = []  
  
    def déposer(self, montant):  
        """Méthode publique pour déposer de l'argent"""  
        if montant > 0:  
            self._solde += montant  
            self.__transactions.append(f"Dépôt: {montant}")  
            print(f"Dépôt de {montant}€ effectué.")  
        else:  
            print("Montant de dépôt invalide.")  
  
    def __afficher_solde(self):  
        """Méthode protégée pour afficher le solde"""  
        print(f"Solde actuel: {self._solde}€")  
  
    def __enregistrer_transaction(self, transaction):  
        """Méthode privée pour enregistrer une transaction"""  
        self.__transactions.append(transaction)  
        print("Transaction enregistrée.")
```

- Créer une instance
- Déposer 100€ dans le compte
- Accéder directement sur l'objet aux méthodes de l'instance
- Accéder via Name Mangling aux méthodes privées
- Afficher le solde

Correction :

```
# -----
# Exercice 2 - classe CompteBancaire
# -----


# Création d'une instance
compte = CompteBancaire("Alice")

# Déposer 100€ dans le compte
compte.deposer(100)

# Tentative d'accès aux méthodes protégées et privées

# compte._afficher_solde()           # Possible mais déconseillé

# compte.__enregistrer_transaction("Retrait: 50€") # Provoque une
AttributeError

# Accès via name mangling
compte._CompteBancaire__enregistrer_transaction("Retrait: 50€")

# Affichage du solde en utilisant une méthode protégée
compte._afficher_solde()
```

Sortie :

```
Dépôt de 100€ effectué.
Transaction enregistrée.
Solde actuel: 100€
```

3- Créer une classe « Rectangle »

```
class Rectangle:
    def __init__(self, largeur, hauteur):
        self._largeur = largeur
        self._hauteur = hauteur

    @property
    def largeur(self):
        return self._largeur

    @largeur.setter
    def largeur(self, valeur):
        if valeur > 0:
            self._largeur = valeur
        else:
            raise ValueError("La largeur doit être
                             positive.")

    @property
    def hauteur(self):
        return self._hauteur

    @hauteur.setter
    def hauteur(self, valeur):
        if valeur > 0:
            self._hauteur = valeur
        else:
            raise ValueError("La hauteur doit être
                             positive.")

    @property
    def aire(self):
        return self._largeur * self._hauteur
```

- Créer une instance
- Accéder aux propriétés et les afficher
- Modifier les propriétés
- Afficher de nouveau les propriétés
- Essayez d'affecter -3 à la propriété `__largeur`

Correction :

```

# -----
# Exercice 2 - classe Rectangle
# -----


# Cr éation d'une instance
rect = Rectangle(5, 10)

# Acc ès aux propriétés
print(rect.largeur)
print(rect.hauteur)
print(rect.aire)

# Modification des propriétés
rect.largeur = 7
rect.hauteur = 12

print(rect.largeur)
print(rect.hauteur)
print(rect.aire)

# Tentative de d éfinition avec une valeur invalide
# rect.largeur = -3 # Provoque une ValueError: La
Largeur doit être positive.

```

Sortie :

```
5  
10  
50  
7  
12  
84
```

V. MEMBRES STATIQUES

En programmation orientée objet (POO), un **membre statique** est un attribut ou une méthode qui appartient à la classe elle-même plutôt qu'à une instance particulière de la classe. Cela signifie que tous les objets (instances) de la classe partagent le même membre statique.

Créer un nouvel environnement Python3 en récupérant le dossier Lib/utils des chapitres précédents.

Créer un dossier « Lib/classes » et dedans créer une classe Voiture

Types de Membres Statiques en Python

En Python, les membres statiques se divisent principalement en deux catégories :

- **Attributs de Classe**

Les **attributs de classe** sont des variables définies directement au sein de la classe, en dehors de toute méthode. Ils sont partagés par toutes les instances de la classe.

```
class Voiture:  
    roues = 4 # Attribut de classe  
  
    def __init__(self, marque, modèle):  
        self.marque = marque # Attribut d'instance  
        self.modèle = modèle # Attribut d'instance|
```

- **Méthodes Statiques**

Les **méthodes statiques** sont des fonctions définies au sein d'une classe qui ne nécessitent pas l'accès aux attributs d'instance (self) ou aux attributs de classe (cls).

Elles sont décorées avec **@staticmethod**.

Créer un classe « Math »

```
b > classes > {} math.py > Math > add
```

```
1 class Math:  
2     @staticmethod  
3     def add(a, b):  
4         return a + b
```

Il est essentiel de distinguer **les méthodes statiques** des **méthodes de classe**, car elles servent des buts différents malgré leur similarité apparente.

Aspect	Méthodes Statiques	Méthodes de Classe
Définition	Fonctions indépendantes, n'accèdent ni à <code>self</code> ni à <code>cls</code>	Fonctions qui reçoivent la classe (<code>cls</code>) comme premier argument
Décorateur	<code>@staticmethod</code>	<code>@classmethod</code>
Accès aux Membres	Aucun accès aux attributs de classe ou d'instance	Accès aux attributs de classe via <code>cls</code>
Utilisation Typique	Fonctions utilitaires ou logiques indépendantes de la classe	Fabrication d'instances ou manipulation des attributs de classe

Utilisation des membres statiques

Attributs de Classe

Les **attributs de classe** sont définis directement dans la classe et sont partagés par toutes les instances. Ils peuvent être accédés via la classe ou via une instance.

```
# Accès via la classe
print(Voiture.roues) # 4

# Accès via une instance
voiture1 = Voiture("Toyota", "Corolla")
print(voiture1.roues) # 4

# Modification via la classe
Voiture.roues = 5
print(voiture1.roues) # 5

# Modification via une instance (crée un
# attribut d'instance qui masque l'attribut de
# classe)
voiture1.roues = 6
print(voiture1.roues)
print(Voiture.roues) # demeure 5

# Modification via la classe après modification
# de l'instance. Si l'instance a masqué l'attribut
# de classe toute modification de l'attribut de
# classe ne sera pas répercuté dans l'instance
Voiture.roues = 7
print(voiture1.roues) # reste à 6
```

```
4
4
5
6
5
6
```

- Méthodes Statiques

-

Les méthodes statiques sont utiles pour définir des fonctions qui logiquement appartiennent à la classe mais qui n'ont pas besoin d'accéder ou de modifier les attributs de classe ou d'instance.

Rajoutons une méthode statique dans la classe Math

```
class Math:  
    @staticmethod  
    def add(a, b):  
        return a + b  
  
    @staticmethod  
    def multiply(a, b):  
        return a * b
```

Puis testons :

```
# Appel via la classe  
print(Math.add(5, 3))      # 8  
print(Math.multiply(5, 3))  # 15  
  
# Appel via une instance  
math_instance = Math()  
print(math_instance.add(2, 4))  # 6  
print(math_instance.multiply(2, 4))# 8
```

```
-----  
8  
15  
6  
8
```

Bonnes Pratiques pour Utiliser les Membres Statiques

1. Utiliser des Attributs de Classe pour les Données Partagées :

- Réservez les attributs de classe pour les informations qui doivent être partagées par toutes les instances.
- Exemple : Constantes, paramètres de configuration, compteurs, etc.

2. Définir des Méthodes Statiques pour les Fonctions Utilitaires :

- Placez les fonctions qui ne nécessitent pas d'accès aux attributs de la classe ou de l'instance en tant que méthodes statiques.
- Cela améliore la lisibilité et l'organisation du code.

3. Limiter l'Utilisation des Méthodes Statiques :

- Ne les utilisez que lorsque cela est logique du point de vue de la conception.
- Évitez de mettre trop de logique dans des méthodes statiques qui pourraient être mieux placées ailleurs.

4. Clarté et Simplicité :

- Gardez les membres statiques simples et cohérents avec la responsabilité de la classe.
- Assurez-vous que leur utilisation est intuitive et bien documentée.

5. Préférer les Méthodes de Classe lorsqu'un Accès à la Classe est Nécessaire :

- Si une méthode doit accéder ou modifier les attributs de classe, utilisez une méthode de classe plutôt qu'une méthode statique.

6. Documentation :

- Documentez clairement les membres statiques pour indiquer leur rôle et leur utilisation prévue.

NB : Python ne dispose pas de classe statique contrairement à de nombreux langages.

Plutôt que d'introduire une classification supplémentaire de classes (comme les classes statiques), Python utilise les **modules** et les **fonctions globales** pour fournir des fonctionnalités similaires à celles des classes statiques dans d'autres langages.

Raisons Principales :

- **Modules comme Espaces de Noms** : Les modules Python servent déjà de conteneurs pour regrouper des fonctions, des variables et des classes, ce qui rend l'introduction des classes statiques redondante.
- **Flexibilité** : Python favorise une approche plus flexible et minimaliste, permettant aux développeurs de choisir la meilleure façon d'organiser leur code sans contraintes supplémentaires.
- **Pas de Besoin Imposé** : Étant donné que les modules peuvent accomplir la même chose que les classes statiques, Python n'impose pas de syntaxe ou de mécanisme supplémentaire.

Cependant, il est toujours possible d'**émuler** les classes statiques en Python si vous le souhaitez, en utilisant des classes avec uniquement des méthodes statiques ou en combinant des classes et des méthodes de classe.

VI. CLASSES STATIQUES (EMULEES)

Bien que Python ne propose pas de classes statiques au sens strict, il est possible d'émuler ce comportement de plusieurs manières. Deux approches principales sont :

1. Utiliser une classe avec uniquement des méthodes statiques et des attributs de classe
2. Utiliser des modules comme conteneurs de fonctions utilitaires

Utiliser une Classe avec Uniquement des Méthodes Statiques

Cette approche consiste à définir une classe contenant uniquement des méthodes statiques et, éventuellement, des attributs de classe.

Vous pouvez également empêcher l'instanciation de la classe en levant une exception dans le constructeur.

Créons une classe MathUtilities qui ne contient que des méthodes statiques.

```
class MathUtilities:  
    @staticmethod  
    def add(a, b):  
        return a + b  
  
    @staticmethod  
    def multiply(a, b):  
        return a * b
```

Empêchons son instantiation. Pour ce faire nous allons utiliser la méthode spéciale `__new__` qui est exécutée lors de l'instanciation

```
class MathUtilities:  
    @staticmethod  
    def add(a, b):  
        return a + b  
  
    @staticmethod  
    def multiply(a, b):  
        return a * b  
  
    # On empêche l'instanciation  
    def __new__(cls, *args, **kwargs):  
        raise TypeError("Cette classe ne peut pas être instanciée.")
```

Testons :

```
# Tentative d'instaciation de La classe statique emulée  
# mathUtilities  
# Cela Leve l'exception TypeError  
math_util = MathUtilities()
```

```
Traceback (most recent call last):
  File "G:\Desktop\tutos\python\vscode\pythonpoo\04-statique\main
<module>
    math_util = MathUtilities()
    ^^^^^^^^^^^^^^

  File "G:\Desktop\tutos\python\vscode\pythonpoo\04-statique\Lib\
sites.py", line 12, in __new__
    raise TypeError("Cette classe ne peut pas être instanciée.")
TypeError: Cette classe ne peut pas être instanciée.
[nodemon] app crashed - waiting for file changes before starting.
```

N.B :

Les méthodes statiques n'accèdent jamais aux attributs de classe ou d'instance. Ils ne peuvent recevoir **cls** ou **self** en paramètre, contrairement aux méthodes de classe marquées par le décorateur **@classmethod**

Utilisation des Modules pour des Fonctionnalités Similaires

Python encourage l'utilisation des **modules** pour regrouper des fonctions utilitaires, ce qui est souvent plus simple et plus efficace que d'utiliser une classe statique.

Dans dossier Lib/utils, créer un fichier nommé `math_utils.py` :

```
> utils > {} math_utils.py > M multiply
1  def add(a, b):
2      return a + b
3
4  def multiply(a, b):
5      return a * b
```

Puis utilisons ce module dans le main.py

```
from Lib.utils import math_utils
```

```
print("-----")
print("utilisation d'un module")
print(math_utils.add(10, 5)) # 15
print(math_utils.multiply(10, 5)) # 50|
```

Sortie :

```
-----
Utilisation d'une classe statique
15
50
-----
utilisation d'un module
15
50
```

Bonnes Pratiques pour Utiliser les Classes Statiques en Python

1. Préférer les Modules pour les Fonctions Utilitaires :

- Si vous avez uniquement des fonctions utilitaires, envisagez de les placer dans un module plutôt que dans une classe statique.

2. Utiliser des Méthodes Statiques Judicieusement :

- Réservez les méthodes statiques aux fonctions qui logiquement appartiennent à la classe mais ne nécessitent pas d'accès aux attributs de classe ou d'instance.

3. Limiter l'Utilisation des Classes Statiques :

- N'utilisez pas de classes statiques pour tout. Utilisez-les uniquement lorsque cela apporte une réelle valeur en termes d'organisation ou de logique.

VII. CLASSES ABSTRAITES

Une classe abstraite est une classe qui **ne peut pas être instanciée** directement et qui sert de **modèle** pour d'autres classes.

Elle définit un ensemble de méthodes **abstraites** que les classes dérivées (ou concrètes) doivent implémenter.

Les classes abstraites permettent de **définir des interfaces** et de **forcer les classes dérivées** à respecter certaines conventions, assurant ainsi une **cohérence** dans le code.

Caractéristiques Clés :

- **Non Instanciable** : Vous ne pouvez pas créer d'objet directement à partir d'une classe abstraite.
- **Méthodes Abstraites** : Méthodes déclarées sans implémentation dans la classe abstraite.
- **Modèle pour les Sous-Classes** : Les classes dérivées doivent implémenter les méthodes abstraites.

En Python, les classes abstraites sont implémentées à l'aide du **module abc** (Abstract Base Classes). Ce module fournit les outils nécessaires pour définir des classes abstraites et des méthodes abstraites.

Le Module abc

Le module abc fait partie de la bibliothèque standard de Python et fournit les fonctionnalités suivantes :

- **ABC** : Une classe de base pour définir des classes abstraites.

- **abstractmethod** : Un décorateur pour déclarer des méthodes abstraites.
- **abstractproperty** : Un décorateur pour déclarer des propriétés abstraites (déconseillé en faveur de `@property` combiné avec `@abstractmethod`).

Créer un nouvel environnement Python3. Et créer une classe nommée « FormeGeometrique ».

Définir une Classe Abstraite

Pour définir une classe abstraite en Python, vous devez hériter de la classe ABC et utiliser le décorateur `@abstractmethod` pour déclarer les méthodes abstraites.

```
Lib > classes > {} formegeometrique.py > ...
1  from abc import ABC, abstractmethod
2
3  class FormeGeometrique(ABC):
4      @abstractmethod
5      def aire(self):
6          pass
7
8      @abstractmethod
9      def perimetre(self):
10         pass
11
12     def greetigns(self):
13         print("merci d'utiliser notre super classe!")
14
```

En héritant de ABC, la classe devient une classe abstraite. Le Décorateur `@abstractmethod` indique que la méthode doit être implémentée par toute classe dérivée. La classe abstraite peut cependant posséder des méthodes non abstraites

Méthodes Abstraites

Les méthodes abstraites sont des méthodes déclarées dans une classe abstraite sans implémentation.

Elles définissent une interface que les classes dérivées doivent respecter.

Pour utiliser une classe abstraite, vous devez créer une **classe concrète** qui hérite de la classe abstraite et implémente toutes les méthodes abstraites.

Dans notre exemple créons une classe Rectangle qui hérite de FormeGeometrique

```
from Lib.classes.formegeometrique import FormeGeometrique

class Rectangle(FormeGeometrique):
    def __init__(self, largeur, hauteur):
        self.largeur = largeur
        self.hauteur = hauteur

    def aire(self):
        return self.largeur * self.hauteur

    def perimetre(self):
        return 2 * (self.largeur + self.hauteur)
```

Ici, la classe abstraite **FormeGeometrique** définit les méthodes abstraites « aire » et « périmètre ».

La classe concrète **Rectangle** : Hérite de **FormeGeometrique** et implémente les méthodes « aire » et « périmètre».

Importons FormeGemetrique et Rectangle dans main.py :

```
from Lib.classes.formegeometrique import FormeGeometrique
from Lib.classes.rectangle import Rectangle
```

```

from Lib.utils import utils
from Lib.classes.formegeometrique import FormeGeometrique
from Lib.classes.rectangle import Rectangle

if __name__ == "__main__":
    #instanciation d'un rectangle
    rect = Rectangle(5,10)
    utils.show(rect, 'rect')

    rect.greetings()

    print("-----")

```

```

Objet observé: rect
Objet de classe: Rectangle

Propriétés de classe:
{'_abc_impl': <_abc._abc_data object at 0x0000026249431540>}

Propriétés d'instance:
{'hauteur': 10, 'largeur': 5}

Méthodes:
['aire', 'greetings', 'perimetre']
merci d'utiliser notre super classe!

```

Restrictions des Classes Abstraites

- **Non Instantiable** : Vous ne pouvez pas créer d'instance directement à partir d'une classe abstraite.

```
print("-----")  
  
# tentative d'intanciation de la classe abstraite:  
# Leve une exception TypeError  
forme = FormeGeometrique()
```

```
-----  
Traceback (most recent call last):  
  File "G:\Desktop\tutos\python\vscode\pythonpoo\05-abstraction\main.py",  
  in <module>  
    forme = FormeGeometrique()  
               ^^^^^^^^^^^^^^^^^^  
TypeError: Can't instantiate abstract class FormeGeometrique with abstract  
aire, perimetre  
[nodemon] app crashed - waiting for file changes before starting...  
█
```

- **Implémentation Obligatoire** : Les classes dérivées doivent implémenter toutes les méthodes abstraites. Sinon, elles resteront abstraites et ne pourront pas être instanciées non plus.

Créons une classe Animal abstraite avec une méthode abstraite « parler »

```
from abc import ABC, abstractmethod  
  
class Animal(ABC):  
  
    @abstractmethod  
    def parler(self):  
        pass
```

Puis créons une classe Chien dans laquelle nous oublions d'implémenter la méthode abstraite de Animal :

```
Lib > classes > chien.py > Chien > marcher
1   from Lib.classes.animal import Animal
2
3   class Chien(Animal):
4       def marcher(self):
5           print("Je tourne en rond...")
```

Puis revenons dans main.py

```
from Lib.classes.chien import Chien

print("-----")
# tentative d'instanciation de la classe Chien
# dans laquelle on a oublié d'implémenter
# la méthode parler de Animal
chien = Chien()

-----
Traceback (most recent call last):
  File "G:\Desktop\tutos\python\vscode\pythonpoo\05-abstract
ion\main.py", line 25, in <module>
    chien = Chien()
              ^
TypeError: Can't instantiate abstract class Chien with abstr
act method parler
[nodemon] app crashed - waiting for file changes before star
ting...
□
```

Getters et Setters Abstraits

On peut également définir dans la classe abstraite des getters et setters abstraits qui devront obligatoirement être implement dans la classe fille.

Créer une classe Employe :

```
classes > {} employe.py > ...
from abc import ABC, abstractmethod

class Employe(ABC):
    @property
    @abstractmethod
    def salaire(self):
        pass

    @salaire.setter
    @abstractmethod
    def salaire(self, valeur):
        pass
```

Créer une classe Ingénieur :

```
from Lib.classes.employe import Employe

class Ingénieur(Employe):
    def __init__(self, salaire):
        self._salaire = salaire

    @property
    def salaire(self):
        return self._salaire

    @salaire.setter
    def salaire(self, valeur):
        if valeur < 0:
            raise ValueError("Le salaire ne peut pas
                            être négatif.")
        self._salaire = valeur
```

Ici toute la classe fille doit implémenter la propriété salaire avec un getter et un setter.

La classe concrète **Ingénieur** implémente la propriété salaire avec des validations.

Testons :

```
from Lib.classes.ingenieur import Ingénieur
```

```
print("-----")
# Test de la classe Ingénieur
mike = Ingenieur(5000)
utils.show(mike, 'mike')

print(f"Le salaire de mike se monte à {mike.salaire} €")
print("mike reçoit une augmentation...")
mike.salaire = 6000
print(f"Le salaire de mike se monte maintenant à {mike.salaire} €")
```

```
Objet observé: mike
Objet de classe: Ingenieur
```

```
Propriétés de classe:
{'_abc_impl': <_abc._abc_data object at 0x000001D6B8442300>,
 'salaire': '<property object>'}
```

```
Propriétés d'instance:
{'_salaire': 5000}
```

```
Méthodes:
```

```
[]
```

```
Le salaire de mike se monte à 5000 €
mike reçoit une augmentation...
Le salaire de mike se monte maintenant à 6000 €
```

VIII. LES INTERFACES

En programmation orientée objet (POO), une **interface** est un contrat qui définit un ensemble de méthodes que les classes doivent implémenter, sans fournir leur implémentation. Les interfaces permettent de spécifier les comportements qu'une classe doit supporter, assurant ainsi une **cohérence** et facilitant l'**interopérabilité** entre différentes parties d'un programme.

Caractéristiques Clés des Interfaces :

- **Abstraction** : Les interfaces ne contiennent que des déclarations de méthodes, sans implémentation.
- **Contrat** : Toute classe qui implémente une interface doit fournir une implémentation pour toutes les méthodes déclarées.
- **Flexibilité** : Permettent de définir des comportements communs sans imposer une hiérarchie de classes rigide.

Implémentation des Interfaces en Python

Python, étant un langage dynamique et flexible, ne dispose pas d'un mot-clé spécifique pour les interfaces. Cependant, il offre plusieurs moyens d'implémenter des interfaces :

Utilisation des Classes Abstraites (abc module)

Le module abc (Abstract Base Classes) de la bibliothèque standard de Python permet de définir des **classes abstraites** et des **méthodes abstraites**, ce qui est l'approche la plus courante pour implémenter des interfaces en Python.

Étapes pour Créer une Interface avec abc :

1. **Hériter de ABC** : Toute classe abstraite doit hériter de abc.ABC.
2. **Définir des Méthodes Abstraites** : Utiliser le décorateur @abstractmethod pour déclarer des méthodes sans implémentation.

The screenshot shows a code editor with a dark theme. At the top, there's a breadcrumb navigation bar: 'classes > {} animal.py > IAnimal > se_deplacer'. Below this, the code for the `IAnimal` interface is displayed:

```
from abc import ABC, abstractmethod

class IAnimal(ABC):
    @abstractmethod
    def parler(self):
        pass

    @abstractmethod
    def se_deplacer(self):
        pass
```

Ici, `IAnimal` est une interface qui définit deux méthodes abstraites : `parler` et `se_deplacer`.

Utilisation des Protocoles (`typing.Protocol`)

Introduits dans le module `typing` à partir de Python 3.8 (et disponibles via le module `typing_extensions` pour les versions antérieures), les **protocoles** permettent de définir des interfaces basées sur le **type** plutôt que sur l'**héritage**.

Cela s'inscrit dans le paradigme de **typage structurel** (ou **duck typing**).

Avantages des Protocoles :

- **Flexibilité** : Les classes n'ont pas besoin d'hériter explicitement d'un protocole pour être considérées comme conformes.
- **Compatibilité avec l'Héritage Multiple** : Permet aux classes d'implémenter plusieurs protocoles sans contrainte.

```
lasses > { printable.py > ...
from typing import Protocol

class IPrintable(Protocol):
    def print(self) -> None:
        """ protocole Printable"""

```

Ici, toute classe qui possède une méthode print sans arguments et qui retourne None sera considérée comme conforme au protocole IPrintable, même si elle n'hérite pas explicitement de IPrintable.

Afin pouvoir le vérifier nous allons rendre le type IPrintable vérifiable durant l'exécution.

```
.../printable.py > ...
from typing import Protocol
from typing import runtime_checkable

@runtime_checkable
class IPrintable(Protocol):
    def print(self) -> None:
        """ protocole Printable"""

```

Puis créons une classe Document qui possède une méthode de même signature que le protocole IPrintable :

```
class Document():
    def print(self) -> None:
        print("I can be printed!!")
```

Puis revenons dans main.py où nous allons vérifier que Document possède le type IPrintable.

```
from Lib.classes.printable import IPrintable
from Lib.classes.document import Document
from Lib.classes.canard import Canard
from Lib.classes.personne import Personne

def is_printable(doc: IPrintable):
    if isinstance(doc, IPrintable):
        print("Document is printable")
```

LA sortie nous montre que c'est bien le cas :

```
[nodemon] starting `python main.py`
Document is printable
[nodemon] clean exit - waiting for ...
```

Duck Typing

Le duck typing est un principe selon lequel la compatibilité est déterminée par la présence de certains comportements (méthodes et attributs) plutôt que par l'héritage ou l'implémentation explicite d'interfaces

Créer un classe Canard et une classe Personne toute deux avec une méthode « parler() »

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

```
classes > perso.py > Personne > parler
└ class Personne:
    └ def parler(self):
        └ print("Bonjour!")
```

Revenons dans main.py

```
from Lib.classes.canard import Canard
from Lib.classes.personne import Personne
```

Définissons une fonction faire_parler() :

```
def faire_parler(obj: object) -> None:
    obj.parler()
```

Puis instancions un canard et une personne :

```
john = Personne()
donald = Canard()

faire_parler(john)
faire_parler(donald)
```

Ici, la fonction faire_parler fonctionne avec tout objet qui possède une méthode parler, sans se soucier de leur type ou de leur hiérarchie.

Différences entre Interfaces en Python et dans d'autres Langages

Comparons rapidement les interfaces en Python avec celles d'autres langages comme Java ou C#.

Aspect	Python	Java/C#
Mécanisme	Classes abstraites (<code>abc</code>), Protocoles (<code>typing.Protocol</code>), Duck Typing	Mots-clés spécifiques (<code>interface</code>), Héritage multiple via interfaces
Héritage	Optionnel pour les Protocoles, obligatoire pour <code>ABC</code>	Obligatoire via le mot-clé <code>implements</code>
Typage	Dynamique (duck typing, typage structurel)	Statique (typage nominal)
Héritage Multiple	Possible via Protocoles	Supporté via les interfaces
Implémentation Obligatoire	Oui pour <code>ABC</code> , Non nécessaire pour Protocoles	Oui pour toutes les méthodes déclarées

Bonnes Pratiques pour Utiliser les Interfaces en Python

1. Utiliser les Classes Abstraites pour les Contrats Stricts :

- Si vous avez besoin d'imposer l'implémentation de certaines méthodes, utilisez des classes abstraites avec abc.

2. Préférer les Protocoles pour la Flexibilité :

- Lorsque l'héritage n'est pas nécessaire ou souhaité, utilisez les protocoles pour définir des interfaces basées sur le type.

3. Documenter les Interfaces :

- Utilisez des **docstrings** pour expliquer les méthodes et les attentes des interfaces, facilitant ainsi leur implémentation par d'autres développeurs.

4. Limiter l'Utilisation des Classes Abstraites :

- Ne les utilisez que lorsque cela apporte une réelle valeur en termes de structuration et de contrôle des comportements.

5. Combiner avec les Propriétés :

- Utilisez des propriétés abstraites pour définir des interfaces qui incluent des attributs avec des getters et setters.

6. Respecter le Principe de Substitution de Liskov (LSP) :

- Assurez-vous que les classes implémentant une interface peuvent être utilisées de manière interchangeable sans altérer le comportement attendu.

7. Éviter les Méthodes Trop Génériques :

- Les méthodes dans les interfaces doivent être spécifiques et avoir un but clair, évitant ainsi les abstractions trop vagues.

IX. LES TRAITS

En **programmation orientée objet (POO)**, un **trait** est un concept permettant de **réutiliser des méthodes** dans différentes classes sans recourir à l'héritage classique. Les traits offrent une **modularité supplémentaire** en permettant d'ajouter des fonctionnalités spécifiques à des classes indépendamment de leur hiérarchie d'héritage.

En d'autres termes, un trait c'est une petite librairie qu'on peut brancher directement à une classe.

Caractéristiques Clés des Traits :

- **Réutilisabilité** : Permettent de réutiliser des méthodes dans plusieurs classes.
- **Indépendance** : Ne dépendent pas de la hiérarchie d'héritage.
- **Composition** : Favorisent la composition de comportements plutôt que l'héritage multiple complexe.

Comparaison avec les Interfaces et les Classes Abstraites :

- **Interfaces** : Définissent des contrats sans implémentation. Les traits, en revanche, fournissent des implémentations concrètes.
- **Classes Abstraites** : Peuvent contenir des méthodes implémentées et non implémentées. Les traits se concentrent principalement sur la fourniture de comportements réutilisables sans imposer une hiérarchie.

Pourquoi Utiliser des Traits ?

L'utilisation des traits présente plusieurs avantages dans la conception logicielle :

1. **Éviter l'Héritage Multiple Complex** : L'héritage multiple peut devenir difficile à gérer et à maintenir. Les traits offrent une alternative plus flexible pour ajouter des fonctionnalités.
2. **Favoriser la Composition** : Encourage la création de classes en combinant des comportements spécifiques plutôt qu'en étendant une hiérarchie d'héritage rigide.
3. **Réduire la Redondance** : Permet de définir des comportements réutilisables une seule fois et de les inclure dans plusieurs classes.
4. **Améliorer la Modularité** : Facilite la maintenance et l'évolution du code en séparant les fonctionnalités en modules distincts.

Implémentation des Traits en Python

Bien que Python ne dispose pas d'un support explicite pour les traits, il est possible de reproduire leur comportement en utilisant des **mixins** et l'**héritage multiple**.

Utilisation des Mixins

Un mixin est une classe conçue pour fournir des fonctionnalités supplémentaires à d'autres classes via l'héritage multiple. Contrairement aux classes principales, les mixins n'ont généralement pas d'état propre (pas d'attributs d'instance) et se concentrent sur des comportements spécifiques.

Caractéristiques des Mixins :

- **Pas d'instanciation propre** : Les mixins ne sont pas destinés à être instanciés directement.
- **Fonctionnalités Spécifiques** : Fournissent des méthodes utilitaires ou des comportements additionnels.
- **Héritage Multiple** : Sont utilisés en combinaison avec d'autres classes via l'héritage multiple.

Héritage Multiple

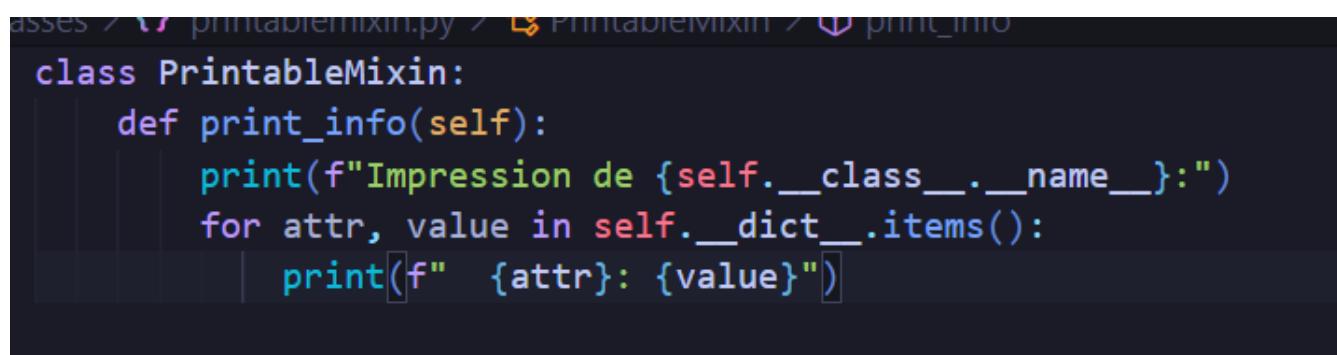
L'**héritage multiple** permet à une classe de hériter de plusieurs classes parent. En combinant des mixins avec l'héritage multiple, il est possible d'ajouter des comportements variés et réutilisables à différentes classes sans créer de hiérarchies complexes.

Considérations :

- **Ordre des Classes Parent** : L'ordre dans lequel les classes parent sont listées influence la résolution des méthodes (MRO – Method Resolution Order).
- **Éviter les Conflits** : Assurez-vous que les mixins n'ont pas de méthodes avec les mêmes noms pour éviter les conflits.

Pratiquons

Créer une classe PrintableMixin



```
ases / printabimixin.py / PrintableMixin / print_info
class PrintableMixin:
    def print_info(self):
        print(f"Impression de {self.__class__.__name__}:")
        for attr, value in self.__dict__.items():
            print(f"  {attr}: {value}")
```

Puis créons une classe Document

```
from Lib.classes.printableMixin import PrintableMixin  
|  
class Document(PrintableMixin):  
    def __init__(self, titre, contenu):  
        self.titre = titre  
        self.contenu = contenu
```

Puis allons dans mains.py

```
from Lib.utils import utils  
from Lib.classes.document import Document  
from Lib.classes.printableMixin import PrintableMixin  
  
if __name__ == "__main__":  
  
    doc = Document("Guide Python", "Contenu du guide.")  
    doc.print_info()
```

```
Impression de Document:  
titre: Guide Python  
contenu: Contenu du guide.  
[nodemon] clean exit - waiting for changes before restart
```

PrintableMixin : Fournit la méthode print_info pour imprimer les informations de l'objet.

Document: Hérite de PrintableMixin pour obtenir la fonctionnalité d'impression sans dupliquer le code.

Avantages : Réutilisation du comportement d'impression dans plusieurs classes sans créer de hiérarchie complexe.

Combinaison de Traits via l'Héritage Multiple

Imaginons une application où nous avons plusieurs comportements à ajouter, tels que l'enregistrement dans un journal et la gestion des erreurs.

Créons deux classes

- JournalMixin
- ErrorhandlingMixin

```
Lib > classes > {} journal mixin.py > ⌂ JournalMixin > ⚙ log
1  class JournalMixin:
2      def log(self, message):
3          print(f"[Journal] {message}")
```

```
sses > {} error handling mixin.py > ...
class ErrorHandlingMixin:
    def handle_error(self, error):
        print(f"[Erreur] {error}")
```

Puis créons une classe qui va utiliser les traits

```
from Lib.classes.journal mixin import JournalMixin
from Lib.classes.error handling mixin import ErrorHandlingMixin

class Service(JournalMixin, ErrorHandlingMixin):
    def execute(self, action):
        try:
            self.log("Action en cours d'exécution.")
            action()
            self.log("Action exécutée avec succès.")
        except Exception as e:
            self.handle_error(e)
```

Puis utilisons cette classe dans main.py

```
from Lib.classes.service import Service

from Lib.classes.journal mixin import JournalMixin
from Lib.classes.errorhandling mixin import ErrorHandlingMixin

class Service(JournalMixin, ErrorHandlingMixin):
    def execute(self, action):
        try:
            self.log("Action en cours d'exécution.")
            action()
            self.log("Action exécutée avec succès.")
        except Exception as e:
            self.handle_error(e)
```

Sortie :

```
Impression de Document:
titre: Guide Python
contenu: Contenu du guide.
[Journal] Action en cours d'exécution.
Action réussie.
[Journal] Action exécutée avec succès.
[Journal] Action en cours d'exécution.
Action échoue.
[Erreur] Une erreur s'est produite.
```

Ici :

JournalMixin : Fournit la méthode log pour enregistrer des messages dans un journal.

ErrorHandlingMixin : Fournit la méthode handle_error pour gérer les erreurs.

Service : Hérite des deux mixins et combine leurs fonctionnalités dans la méthode execute.

Bonnes Pratiques pour Utiliser les Traits en Python

1. Créer des Mixins Cohérents :

- Chaque mixin doit avoir une responsabilité claire et spécifique.
- Évitez de surcharger un mixin avec trop de fonctionnalités.

2. Nommer les Mixins Appropriatement :

- Utilisez des suffixes comme Mixin pour indiquer clairement qu'il s'agit de mixins.
- Exemple : PrintableMixin, SerializableMixin.

3. Préférer l'Héritage Multiple Contrôlé :

- Utilisez l'héritage multiple de manière judicieuse pour éviter les conflits de noms et les complexités inutiles.
- Assurez-vous que les mixins n'introduisent pas de dépendances cachées.

Réflexion durant quelques secondes

Bien sûr ! Explorons en détail le concept des **traits** en Python. Bien que Python ne dispose pas d'un mécanisme natif appelé "traits" comme certains autres langages orientés objet (par exemple, Scala ou PHP), il offre des moyens puissants d'implémenter des fonctionnalités similaires grâce à des concepts tels que les **mixins** et l'**héritage multiple**. Cette explication couvrira :

1. Qu'est-ce qu'un Trait ?
2. Pourquoi Utiliser des Traits ?
3. Implémentation des Traits en Python
 - Utilisation des Mixins
 - Héritage Multiple
4. Exemples Illustratifs
 - Exemple 1 : Trait avec un Mixin
 - Exemple 2 : Combinaison de Traits via l'Héritage Multiple
5. Bonnes Pratiques pour Utiliser les Traits en Python
6. Limitations et Alternatives
7. Conclusion

Qu'est-ce qu'un Trait ?

En **programmation orientée objet (POO)**, un **trait** est un concept permettant de **réutiliser des méthodes** dans différentes classes sans recourir à l'héritage classique. Les traits offrent une **modularité** supplémentaire en permettant d'ajouter des fonctionnalités

spécifiques à des classes indépendamment de leur hiérarchie d'héritage.

Caractéristiques Clés des Traits :

- **Réutilisabilité** : Permettent de réutiliser des méthodes dans plusieurs classes.
- **Indépendance** : Ne dépendent pas de la hiérarchie d'héritage.
- **Composition** : Favorisent la composition de comportements plutôt que l'héritage multiple complexe.

Comparaison avec les Interfaces et les Classes Abstraites :

- **Interfaces** : Définissent des contrats sans implémentation. Les traits, en revanche, fournissent des implémentations concrètes.
- **Classes Abstraites** : Peuvent contenir des méthodes implémentées et non implémentées. Les traits se concentrent principalement sur la fourniture de comportements réutilisables sans imposer une hiérarchie.

Pourquoi Utiliser des Traits ?

L'utilisation des traits présente plusieurs avantages dans la conception logicielle :

1. **Éviter l'Héritage Multiple Complex** : L'héritage multiple peut devenir difficile à gérer et à maintenir. Les traits offrent une alternative plus flexible pour ajouter des fonctionnalités.
2. **Favoriser la Composition** : Encourage la création de classes en combinant des comportements spécifiques plutôt qu'en étendant une hiérarchie d'héritage rigide.
3. **Réduire la Redondance** : Permet de définir des comportements réutilisables une seule fois et de les inclure dans plusieurs classes.
4. **Améliorer la Modularité** : Facilite la maintenance et l'évolution du code en séparant les fonctionnalités en modules distincts.

Implémentation des Traits en Python

Bien que Python ne dispose pas d'un support explicite pour les traits, il est possible de reproduire leur comportement en utilisant des **mixins** et l'**héritage multiple**.

Utilisation des Mixins

Un **mixin** est une classe conçue pour fournir des fonctionnalités supplémentaires à d'autres classes via l'héritage multiple.

Contrairement aux classes principales, les mixins n'ont généralement pas d'état propre (pas d'attributs d'instance) et se concentrent sur des comportements spécifiques.

Caractéristiques des Mixins :

- **Pas d'instanciation propre** : Les mixins ne sont pas destinés à être instanciés directement.

- **Fonctionnalités Spécifiques** : Fournissent des méthodes utilitaires ou des comportements additionnels.
- **Héritage Multiple** : Sont utilisés en combinaison avec d'autres classes via l'héritage multiple.

Héritage Multiple

L'**héritage multiple** permet à une classe de hériter de plusieurs classes parent. En combinant des mixins avec l'héritage multiple, il est possible d'ajouter des comportements variés et réutilisables à différentes classes sans créer de hiérarchies complexes.

Considérations :

- **Ordre des Classes Parent** : L'ordre dans lequel les classes parent sont listées influence la résolution des méthodes (MRO – Method Resolution Order).
- **Éviter les Conflits** : Assurez-vous que les mixins n'ont pas de méthodes avec les mêmes noms pour éviter les conflits.

Exemples Illustratifs

Exemple 1 : Trait avec un Mixin

Supposons que nous souhaitions ajouter des fonctionnalités d'impression à plusieurs classes différentes (par exemple, Document, Image, etc.).

python

Copier le code

```
# Définition du Mixin
class PrintableMixin:
    def print_info(self):
        print(f"Impression de {self.__class__.__name__}:")
        for attr, value in self.__dict__.items():
            print(f"  {attr}: {value}")

# Classe principale Document
class Document(PrintableMixin):
    def __init__(self, titre, contenu):
        self.titre = titre
        self.contenu = contenu

# Classe principale Image
class Image(PrintableMixin):
    def __init__(self, chemin, resolution):
        self.chemin = chemin
        self.resolution = resolution

# Utilisation
doc = Document("Guide Python", "Contenu du guide.")
```

```
img = Image("/images/python.png", "1920x1080")

doc.print_info()
# Output:
# Impression de Document:
#   titre: Guide Python
#   contenu: Contenu du guide.
```

```
img.print_info()
# Output:
# Impression de Image:
#   chemin: /images/python.png
#   resolution: 1920x1080
```

Explications :

- **PrintableMixin** : Fournit la méthode `print_info` pour imprimer les informations de l'objet.
- **Document et Image** : Héritent de `PrintableMixin` pour obtenir la fonctionnalité d'impression sans dupliquer le code.
- **Avantages** : Réutilisation du comportement d'impression dans plusieurs classes sans créer de hiérarchie complexe.

Exemple 2 : Combinaison de Traits via l'Héritage Multiple

Imaginons une application où nous avons plusieurs comportements à ajouter, tels que l'enregistrement dans un journal et la gestion des erreurs.

```
python
Copier le code
# Définition des Mixins
class JournalMixin:
    def log(self, message):
        print(f"[Journal] {message}")

class ErrorHandlingMixin:
    def handle_error(self, error):
        print(f"[Erreur] {error}")

# Classe principale qui combine les Mixins
class Service(JournalMixin, ErrorHandlingMixin):
    def execute(self, action):
        try:
            self.log("Action en cours d'exécution.")
            action()
            self.log("Action exécutée avec succès.")
        except Exception as e:
            self.handle_error(e)
```

```

# Utilisation
def action_reussie():
    print("Action réussie.")

def action_échouée():
    print("Action échoue.")
    raise ValueError("Une erreur s'est produite.")

service = Service()
service.execute(action_reussie)
# Output:
# [Journal] Action en cours d'exécution.
# Action réussie.
# [Journal] Action exécutée avec succès.

service.execute(action_échouée)
# Output:
# [Journal] Action en cours d'exécution.
# Action échoue.
# [Erreur] Une erreur s'est produite.

```

Explications :

- **JournalMixin** : Fournit la méthode `log` pour enregistrer des messages dans un journal.
- **ErrorHandlingMixin** : Fournit la méthode `handle_error` pour gérer les erreurs.
- **Service** : Hérite des deux mixins et combine leurs fonctionnalités dans la méthode `execute`.
- **Avantages** : Permet de combiner plusieurs comportements sans créer une hiérarchie d'héritage complexe, favorisant la modularité et la réutilisabilité.

Bonnes Pratiques pour Utiliser les Traits en Python

1. Créer des Mixins Cohérents :

- Chaque mixin doit avoir une responsabilité claire et spécifique.
- Évitez de surcharger un mixin avec trop de fonctionnalités.

2. Nommer les Mixins Appropriatement :

- Utilisez des suffixes comme `Mixin` pour indiquer clairement qu'il s'agit de mixins.
- Exemple : `PrintableMixin`, `SerializableMixin`.

3. Préférer l'Héritage Multiple Contrôlé :

- Utilisez l'héritage multiple de manière judicieuse pour éviter les conflits de noms et les complexités inutiles.

- Assurez-vous que les mixins n'introduisent pas de dépendances cachées.

4. Documenter les Mixins :

- Fournissez des docstrings clairs pour expliquer la fonctionnalité et les méthodes fournies par le mixin.
- Indiquez les prérequis éventuels (par exemple, les attributs que le mixin attend d'être présents dans la classe principale).

5. Utiliser des Méthodes d'Instance Appropriées :

- Les mixins doivent généralement fournir des méthodes d'instance, mais éviter de définir des attributs d'instance à moins que cela ne soit absolument nécessaire.

6. Éviter les Effets de Bord :

- Les mixins ne doivent pas modifier l'état de la classe principale de manière inattendue.
- Ils doivent être prévisibles et transparents dans leur fonctionnement.

7. Tester les Mixins Indépendamment :

- Assurez-vous que les mixins fonctionnent correctement lorsqu'ils sont utilisés avec différentes classes principales.
- Écrivez des tests unitaires pour chaque mixin afin de valider leur comportement.

Limitations des Traits en Python

1. Absence de Support Natif :

- Python ne dispose pas d'un mécanisme natif pour les traits, ce qui peut rendre leur implémentation moins intuitive pour ceux habitués à d'autres langages.

2. Complexité de l'Héritage Multiple :

- L'utilisation excessive de l'héritage multiple peut conduire à des hiérarchies de classes complexes et difficiles à maintenir.
- Les conflits de noms et la résolution des méthodes (MRO) peuvent poser des problèmes.

3. Manque de Contrats Stricts :

- Contrairement aux interfaces dans des langages comme Java, les mixins en Python ne fournissent pas de contrats stricts, ce qui peut mener à des erreurs silencieuses si les méthodes attendues ne sont pas implémentées.

Alternatives aux Traits en Python

1. Modules Utilitaires :

- Utilisez des modules pour regrouper des fonctions utilitaires qui peuvent être importées et utilisées dans différentes classes sans recourir à l'héritage multiple.

```
# math_utils.py

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

```
# utilisation

import math_utils

print(math_utils.add(2, 3))      # Output: 5
print(math_utils.multiply(2, 3)) # Output: 6
```

2. Composition d'Objets :

- Plutôt que d'utiliser l'héritage multiple, composez des objets en les incluant comme attributs dans d'autres classes pour partager des comportements.

```
class Logger:
    def log(self, message):
        print(f"[LOG] {message}")

class Service:
    def __init__(self):
        self.logger = Logger()

    def execute(self):
        self.logger.log("Service exécuté.")
```

3. Protocoles et Typage Structurel :

- Utilisez les **protocoles** du module typing pour définir des interfaces basées sur le type, permettant une vérification statique des types sans nécessiter d'héritage explicite.

```
from typing import Protocol

class Printable(Protocol):
    def print_info(self) -> None:
        ...

class Document:
    def __init__(self, titre):
        self.titre = titre

    def print_info(self) -> None:
        print(f"Document: {self.titre}")

def afficher(obj: Printable):
    obj.print_info()

doc = Document("Guide Python")
afficher(doc) # Output: Document: Guide Python
```

X. PATRONS DE CONCEPTION

design patterns (motifs de conception)

1. Motifs de création
2. Motifs structurels
3. Motifs comportementaux

Patrons de création

Le singleton

Le patron Singleton est un **patron de conception de création** qui garantit qu'une classe n'a qu'une seule instance tout en fournissant un point d'accès global à cette instance. Cela signifie que, peu importe le nombre de fois où la classe est instanciée, seule une seule et unique instance sera créée et partagée partout dans le programme.

Imaginez une application où plusieurs parties du code ont besoin d'accéder à une connexion unique à une base de données. Utiliser un Singleton garantit que toutes ces parties utilisent la même connexion, évitant ainsi les conflits et les surcharges inutiles.

Le Singleton est utilisé dans des situations où une seule instance d'une classe est nécessaire pour contrôler certaines ressources ou assurer une cohérence à travers l'application.

Voici quelques raisons d'utiliser ce patron :

1. Gestion des Ressources Partagées :

- o Connexions à une base de données
- o Fichiers de configuration
- o Gestionnaires de log

2. Contrôle de l'État Global :

- Maintenir un état partagé ou des données globales

3. Faciliter la Communication :

- Fournir un point centralisé pour la communication entre différentes parties de l'application

4. Réduire l'Overhead :

- Éviter la création multiple d'instances coûteuses en ressources

Exemples Typiques :

- **Logger** : Une seule instance qui gère l'écriture des logs de l'application.
- **Gestionnaire de Configuration** : Une instance qui charge et fournit les paramètres de configuration.
- **Gestionnaire de Connexion** : Une instance qui gère la connexion à une base de données.

Implémentation du Singleton en Python

Python, en tant que langage flexible, offre plusieurs façons d'implémenter le Singleton.

Méthode 1 : Utilisation d'une Variable de Classe

Cette méthode consiste à utiliser une variable de classe pour stocker l'instance unique et contrôler son instanciation.

Créons une classe Singleton

```
classes > {} singleton.py > Singleton > __init__  
class Singleton:  
    _instance = None # Variable de classe pour stocker l'instance unique  
  
    def __new__(cls, *args, **kwargs):  
        if cls._instance is None:  
            cls._instance = super(Singleton, cls).__new__(cls)  
            # Initialisation unique ici si nécessaire  
        return cls._instance  
  
    def __init__(self, value):  
        self.value = value
```

Ici :

_instance : Variable de classe privée qui stocke l'unique instance.
__new__ : Méthode spéciale responsable de la création de l'instance.
Elle vérifie si une instance existe déjà et, si ce n'est pas le cas,
en crée une nouvelle.
__init__ : Méthode d'initialisation qui peut être appelée plusieurs
fois, mais l'instance unique est utilisée.

Testons dans main.py

```
from Lib.utils import utils  
from Lib.classes.singleton import Singleton  
  
  
if __name__ == "__main__":  
  
    sing = Singleton(1)  
    sing2 = Singleton(2)  
  
    utils.show(sing, 'sing')  
    utils.show(sing2, 'sing2')
```

```
Objet observé: sing
Objet de classe: Singleton

Propriétés de classe:
{'_instance': <Lib.classes.singleton.Singleton object at 0x000001AE8EC75A10>}

Propriétés d'instance:
{'value': 2}

Méthodes:
[]

Objet observé: sing2
Objet de classe: Singleton

Propriétés de classe:
{'_instance': <Lib.classes.singleton.Singleton object at 0x000001AE8EC75A10>}

Propriétés d'instance:
{'value': 2}

Méthodes:
[]
```

Les deux variables `sing` et `sing2` pointent vers la même instance en mémoire. La création de la seconde classe a modifié la valeur de la propriété `value` pour les deux instances.

`_instance` : Variable de classe privée qui stocke l'unique instance.

`__new__` : Méthode spéciale responsable de la création de l'instance. Elle vérifie si une instance existe déjà et, si ce n'est pas le cas, en crée une nouvelle.

`__init__` : Méthode d'initialisation qui peut être appelée plusieurs fois, mais l'instance unique est utilisée.

Méthode 2 : Utilisation d'un Métaclasse

Les métaclasses permettent de contrôler la création des classes. En définissant une métaclass Singleton, vous pouvez garantir qu'une seule instance d'une classe est créée.

Créons une metaclass :

```
class SingletonMeta(type):
    _instances = {} # Dictionnaire pour stocker les instances

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            # Crée une nouvelle instance si elle n'existe pas
            instance = super(SingletonMeta, cls).__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]
```

Puis une classe SingletonFromMeta :

```
classes > {} singletonfrommeta.py > ...
from Lib.classes.singletonmeta import SingletonMeta

class SingletonFromMeta(metaclass=SingletonMeta):
    def __init__(self, value):
        self.value = value
```

Testons dans main.py

```
print("-----")
singFromMeta = SingletonFromMeta(1)
singFromMeta2 = SingletonFromMeta(2)

utils.show(singFromMeta, 'singFromMeta')
utils.show(singFromMeta2, 'singFromMeta2')
```

```
-----  
Objet observé: singFromMeta  
Objet de classe: SingletonFromMeta  
  
Propriétés d'instance:  
{'value': 1}  
  
Méthodes:  
[]  
Objet observé: singFromMeta2  
Objet de classe: SingletonFromMeta  
  
Propriétés d'instance:  
{'value': 1}  
  
Méthodes:  
[]
```

La création de la seconde instance n'a pas modifié la première création du singleton.

SingletonMeta : Métaclasse qui override la méthode `__call__` pour contrôler la création des instances.

`_instances` : Dictionnaire stockant les instances uniques de chaque classe utilisant cette métaclasse.

Classe Singleton : Classe utilisant `SingletonMeta` comme métaclasse, assurant une seule instance.

Méthode 3 : Utilisation d'un Décorateur

Un décorateur peut être utilisé pour transformer une classe en Singleton en contrôlant son instantiation.

Créons une fonction SingletonDecorator

```
classes > {} singletondecorator.py > ...
def singleton_decorator(cls):
    def get_instance(*args, **kwargs):
        instances[cls] = cls(*args,
                             **kwargs)
        return instances[cls]

    return get_instance
```

Puis une classe SingletonFromDecorator

```
classes > {} singletonfromdecorator.py > ...
from Lib.classes.singletondecorator import singleton_decorator

@singleton_decorator
class SingletonFromDecorator:
    def __init__(self, value):
        self.value = value
```

Puis testons dans main.py

```
from Lib.classes.singletonfromdecorator import SingletonFromDecorator
```

```
print("-----")
singFromDecorator = SingletonFromDecorator(1)
singFromDecorator2 = SingletonFromDecorator(2)

utils.show(singFromDecorator, 'singFromDecorator')
utils.show(singFromDecorator2, 'singFromDecorator2')
```

```
-----
Objet observé: singFromDecorator
Objet de classe: SingletonFromDecorator

Propriétés d'instance:
{'value': 1}

Méthodes:
[]

Objet observé: singFromDecorator2
Objet de classe: SingletonFromDecorator

Propriétés d'instance:
{'value': 1}

Méthodes:
[]
```

singleton : Décorateur qui maintient un dictionnaire des instances.

get_instance : Fonction interne qui retourne l'instance existante ou en crée une nouvelle.

@singleton : Applique le décorateur à la classe Singleton.

Méthode 4 : Utilisation de Modules Python

En Python, les modules sont intrinsèquement des Singletons. En plaçant les variables et les fonctions dans un module, vous pouvez les utiliser comme une instance unique.

Dans Lib/utils créer un fichier nommé singletonmodule.py

```
b > utils > {} singleton_module.py > ...
1  √ class Singleton:
2  √     |   def __init__(self, value):
3  √     |       self.value = value
4
5      singleton_instance = Singleton("Valeur Initiale")
```

Testons dans main.py

```
from Lib.utils import singletonmodule

print("-----")
print(singletonmodule.singleton_instance.value)
singletonmodule.singleton_instance.value = "Nouvelle Valeur"
print(singletonmodule.singleton_instance.value)
```

```
-----
Valeur Initiale
Nouvelle Valeur
```

Module : Le module singletonmodule contient une instance unique singleton_instance.

Accès Global : En important le module, toutes les parties du code accèdent à la même instance.

Méthode 5 : Le Patron Borg

Le Patron Borg (ou Monostate) permet à toutes les instances d'une classe de partager le même état, plutôt que de restreindre l'instanciation à une seule instance.

Créons une classe Borg

```
Lib > classes > {} borg.py > Borg > _init_
1 class Borg:
2     _shared_state = {} # Dictionnaire
3             partagé
4
5     def __init__(self):
6         self.__dict__ = self._shared_state
```

Puis une classe SingletonFromBorg

```
Lib > classes > {} singletonfromborg.py > ...
1 from Lib.classes.borg import Borg
2
3 class SingletonFromBorg(Borg):
4     def __init__(self, value):
5         super().__init__()
6         self.value = value
7
8
```

Puis testons dans main.py

```
from Lib.classes.singletonfromborg import SingletonFromBorg|
```

```
print("-----")
singletonFromBorg = SingletonFromBorg("Premier")
singletonFromBorg2 = SingletonFromBorg("Second")

utils.show(singletonFromBorg, 'singletonFromBorg')
utils.show(singletonFromBorg2, 'singletonFromBorg2')
```

```
Objet observé: singletonFromBorg
Objet de classe: SingletonFromBorg
```

```
Propriétés d'instance:
{'value': 'Second'}
```

```
Méthodes:
```

```
[]
```

```
Objet observé: singletonFromBorg2
Objet de classe: SingletonFromBorg
```

```
Propriétés d'instance:
{'value': 'Second'}
```

```
Méthodes:
```

```
[]
```

Comparaison des Méthodes

Méthode	Avantages	Inconvénients
Variable de Classe	Simple à implémenter, contrôle centralisé	<code>__init__</code> peut être appelé plusieurs fois, gestion manuelle nécessaire
Métaclasse	Approche propre, réutilisable, permet de créer des Singletons multiples	Plus complexe à comprendre et à implémenter
Décorateur	Flexibilité, peut être appliqué à n'importe quelle classe	Peut compliquer la compréhension du code
Module Python	Simplicité, naturellement Singleton, aligné avec les idiomes Python	Limité à un seul module, moins flexible si plusieurs Singletons sont nécessaires
Patrón Borg	Permet plusieurs instances avec un état partagé	Peut être déroutant, ne restreint pas l'instanciation

Avantages

1. Contrôle de l'Instanciation :

- Garantit qu'une seule instance de la classe existe.

2. Point d'Accès Global :

- Facilite l'accès à l'instance unique depuis n'importe quelle partie du code.

3. Gestion des Ressources Partagées :

- Utile pour gérer des ressources coûteuses ou partagées, comme les connexions à une base de données.

4. Encapsulation :

- Cache les détails d'instanciation, simplifiant l'utilisation de la classe.

Bonnes Pratiques

1. Limiter l'Utilisation des Singletons :

- Utilisez-les uniquement lorsque c'est nécessaire. Les Singletons peuvent introduire des complexités supplémentaires.

2. Implémenter des Interfaces Claires :

- Si vous utilisez des Singlenton, définissez des interfaces ou des abstractions pour faciliter le test et réduire le couplage.

3. Gérer l'État avec Précaution :

- Évitez de stocker des états complexes ou mutables dans les Singlentons pour réduire les risques d'erreurs.

4. Faciliter le Test :

- Fournissez des moyens de réinitialiser ou de remplacer l'instance unique lors des tests unitaires.

5. Documenter l'Utilisation :

- Clarifiez l'objectif et l'utilisation du Singleton pour éviter les abus et les malentendus.

La Factory

Le patron de conception **Factory** est un **patron de création** qui fournit une interface pour créer des objets dans une superclasse, mais permet aux sous-classes de modifier le type d'objets qui seront créés.

En d'autres termes, il définit une méthode pour créer des objets, laissant le choix de la classe concrète à instancier aux sous-classes.

Pour résumer, la Factory permet de :

- Encapsuler la logique de création d'objets.
- Dissocier le code client de la création d'objets

On la préfèrera pour la Création de Différents Types d'Objets Basés sur des Paramètres. Par exemple, une application graphique qui peut créer différents types de boutons (WindowsButton, MacButton) en fonction du système d'exploitation.

On l'utilisera également pour la Gestion des Ressources Partagées ou Coûteuses. Par exemple, une connexion à une base de données où une seule instance est nécessaire.

Types de Patrons Factory

Il existe plusieurs variantes du patron Factory, chacune adaptée à des besoins spécifiques. Les principales sont :

1. **Simple Factory**
2. **Factory Method**
3. **Abstract Factory**

1. Simple Factory

Simple Factory n'est pas officiellement un patron de conception reconnu par le livre *Design Patterns* de Gamma et al., mais il est souvent utilisé pour introduire les concepts de base des patrons Factory.

Comment fonctionne la Simple Factory ?

Une classe (souvent appelée Factory) possède une méthode qui crée et retourne des objets de différentes classes basées sur des paramètres fournis.

2. Factory Method

Factory Method est un patron de conception plus formel qui définit une interface pour créer un objet, mais laisse les sous-classes décider de la classe à instancier.

Comment fonctionne la Factory Method?

Elle définit une méthode abstraite dans une classe de base que les sous-classes doivent implémenter pour créer des objets spécifiques.

3. Abstract Factory

Abstract Factory est une extension du patron Factory Method, permettant de créer des familles d'objets liés sans spécifier leurs classes concrètes.

Comment fonctionne l' Abstract Factory ?

Elle fournit une interface pour créer des groupes d'objets liés ou dépendants sans avoir à spécifier leurs classes concrètes.

Implémentation en Python

Voyons comment implémenter chaque type de patron Factory en Python avec des exemples concrets.

1. Simple Factory

Une classe SimpleFactory possède une méthode statique pour créer des objets basés sur des paramètres.

Créons 4 classes : Chien, Chat, Zozio, AnimalFactory

```
classes > {} chien.py > ...
```

```
class Chien:  
    def parler(self):  
        return "Woof!"
```

```
classes > {} chat.py > ...
```

```
class Chat:  
    def parler(self):  
        return "Meow!"
```

```
classes > {} zozio.py > ...
```

```
class Zozio:  
    def parler(self):  
        return "Tweet!"
```

Et maintenant l'AnimalFactory

```
classes > {} animalfactory.py > ...
```

```
from Lib.classes.chien import Chien  
from Lib.classes.chat import Chat  
from Lib.classes.zozio import Zozio  
  
class AnimalFactory:  
    @staticmethod  
    def create_animal(animal_type):  
        if animal_type == "dog":  
            return Chien()  
        elif animal_type == "cat":  
            return Chat()  
        elif animal_type == "bird":  
            return Zozio()  
        else:  
            raise ValueError(f"Unknown animal type: {animal_type}")
```

Et utilisons la dans main.py

```
from Lib.utils import utils
from Lib.classes.animalfactory import AnimalFactory

if __name__ == "__main__":
    print("\nUtilisation de la Simple Factory")

    factory = AnimalFactory()

    chien = factory.create_animal("dog")
    print(chien.parler())

    chat = factory.create_animal("cat")
    print(chat.parler())

    bird = factory.create_animal("bird")
    print(bird.parler())

    print("-----")
```

```
Utilisation de la Simple Factory
Woof!
Meow!
Tweet!
-----
```

2. Factory Method

On définira une interface pour créer un objet, mais laisser les sous-classes décider quelle classe concrète instancier.

Créons une classe abstraite Document et 2 filles PDFDocument et WordDocument

```
from abc import ABC, abstractmethod

class Document(ABC):
    @abstractmethod
    def open(self):
        pass
```

```
> classes > {} pdfdocument.py > ...
from Lib.classes.document import Document

class PDFDocument(Document):
    def open(self):
        print("Opening PDF document.")
```

```
> classes > {} worddocument.py > ...
from Lib.classes.document import Document

class WordDocument(Document):
    def open(self):
        print("Opening Word document.")
```

Créons maintenant une classe de base pour la Factory qui servira de type commun pour que les classes fille puisse être utilisée par des fonctions dans le script principal.

```
from abc import ABC, abstractmethod
from Lib.classes.document import Document

class DocumentFactory(ABC):
    @abstractmethod
    def create_document(self) -> Document:
        pass
```

Puis créons 2 classes fille de cette factory abstraite :

- PDFFactory
- WordFactory

```
classes > {} pdffactory.py > ...
from Lib.classes.documentfactory import DocumentFactory
from Lib.classes.document import Document
from Lib.classes.pdfdocument import PDFDocument

class PDFFactory(DocumentFactory):
    def create_document(self) -> Document:
        return PDFDocument()
```

```
classes > {} wordfactory.py > ...
from Lib.classes.documentfactory import DocumentFactory
from Lib.classes.document import Document
from Lib.classes.worddocument import WordDocument

class WordFactory(DocumentFactory):
    def create_document(self) -> Document:
        return WordDocument()
```

La classe abstraite DocumentFactory permet de donner un **type commun** à toutes ses filles. Nous pourrons donc créer une fonction qui pourra prendre en charge ce type.

Allons dans main.py :

```
from Lib.classes.documentfactory import DocumentFactory

def open_document(factory: DocumentFactory):
    doc = factory.create_document()
    doc.open()

if __name__ == "__main__":
```

Puis utilisons les factory

```
from Lib.classes.pdffactory import PDFFactory
from Lib.classes.wordfactory import WordFactory
```

```
print("-----")
```

```
print("\nUtilisation de la Factory Method")
pdf_factory = PDFFactory()
open_document(pdf_factory)
```

```
word_factory = WordFactory()
open_document(word_factory)
```

```
-----
Utilisation de la Factory Method
Opening PDF document.
Opening Word document.
```

On passe une factory à la fonction. La méthode `create_document()` de la factory retourne un objet du format de sa spécialisation (la `PDFFactory` retourne un `PDFDocument` qui possède également le type `Document`. Puis on appelle la méthode `open()` de l'objet `Document` créé.

Abstract Factory

Avec cette technique on va fournir une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes. Dans la méthode précédente on a créé des factory spécialisée dans un seul type d'objet. Ici la factory pourra créer plusieurs types d'objets.

Créons les classes abstraites des objets qui seront créé dans un fichier nommé `éléments.py`

```
from abc import ABC, abstractmethod

# Interfaces des Produits
class Button(ABC):
    @abstractmethod
    def paint(self):
        pass

class Checkbox(ABC):
    @abstractmethod
    def paint(self):
        pass
```

```

# Produits Concrets pour Windows
class WindowsButton(Button):
    def paint(self):
        print("Dessiner un bouton dans le style Windows.")

class WindowsCheckbox(Checkbox):
    def paint(self):
        print("Dessiner une case à cocher dans le style Windows.")

```

```

# Produits Concrets pour Mac
class MacButton(Button):
    def paint(self):
        print("Dessiner un bouton dans le style Mac.")

class MacCheckbox(Checkbox):
    def paint(self):
        print("Dessiner une case à cocher dans le style Mac.")

```

Puis dans un autre fichier nommé guifactories.py

```

from abc import ABC, abstractmethod
from Lib.classes.elements import WindowsButton, WindowsCheckbox,
MacButton, MacCheckbox, Button, Checkbox

# Interface Abstract Factory
class GUIFactory(ABC):
    @abstractmethod
    def create_button(self) -> Button:
        pass

    @abstractmethod
    def create_checkbox(self) -> Checkbox:
        pass

```

```
# Factories Concrètes
class WindowsFactory(GUIFactory):
    def create_button(self) -> Button:
        return WindowsButton()

    def create_checkbox(self) -> Checkbox:
        return WindowsCheckbox()

class MacFactory(GUIFactory):
    def create_button(self) -> Button:
        return MacButton()

    def create_checkbox(self) -> Checkbox:
        return MacCheckbox()
```

Puis dans main.py créons une méthode à laquelle on passera une factory afin quelle crée et retourne plusieurs objets.

```
from Lib.classes.guifactories import GUIFactory, WindowsFactory, MacFactory
```

```
def paint_gui(factory: GUIFactory):
    button = factory.create_button()
    checkbox = factory.create_checkbox()
    button.paint()
    checkbox.paint()
```

Puis utilisons cette méthode :

```
print("-----")  
  
print("\nUtilisation de l'Abstract Factory")  
  
os_type = input("Sur quelle plateforme ? (windows/mac): ").strip().lower()  
if os_type == "windows":  
    factory = WindowsFactory()  
elif os_type == "mac":  
    factory = MacFactory()  
else:  
    raise ValueError("Plateforme inconnue.")  
  
paint_gui(factory)
```

```
Utilisation de l'Abstract Factory  
Sur quelle plateforme ? (windows/mac): windows
```

```
Utilisation de l'Abstract Factory  
Sur quelle plateforme ? (windows/mac): windows  
Dessiner un bouton dans le style Windows.  
Dessiner une case à cocher dans le style Windows.
```

Comparaison des Types de Patrons Factory

Type de Factory	Description	Avantages	Inconvénients
Simple Factory	Une seule classe <code>Factory</code> qui crée différents objets basés sur des paramètres.	Simple à implémenter.	Peut devenir complexe avec l'ajout de nouveaux types d'objets.
Factory Method	Définition d'une méthode abstraite dans une classe de base que les sous-classes doivent implémenter pour créer des objets.	Favorise l'extensibilité et le respect du principe de responsabilité unique.	Peut nécessiter la création de nombreuses sous-classes.
Abstract Factory	Fournit une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes.	Facilite la création de familles d'objets cohérentes.	Complexité accrue avec plusieurs familles de produits.

Le Builder

Le patron de conception Builder est un patron de création qui permet de séparer la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction puisse créer différentes représentations.

Structure

Le patron Builder se compose de plusieurs composants clés qui travaillent ensemble pour construire un objet complexe.

- Le Builder
- Le ConcreteBuilder
- Le Director
- Le Product

• Le Product

Le product est l'objet que le Builder devra construire progressivement.

Créer une classe Car :

```
class Car:  
    def __init__(self):  
        self.make = None  
        self.model = None  
        self.year = None  
        self.color = None  
  
    def describe(self):  
        print(f"Car: {self.year} {self.make} {self.model} in {self.color}")
```

• Le Builder (Classe Abstraite)

Le Builder est l'interface qui définit les actions du Builder.

Créons une classe CarBuilder

```
classes > {} carbuilder.py > ...
from abc import ABC, abstractmethod
from Lib.classes.car import Car

class CarBuilder(ABC):
    @abstractmethod
    def set_make(self, make: str):
        pass

    @abstractmethod
    def set_model(self, model: str):
        pass

    @abstractmethod
    def set_year(self, year: int):
        pass

    @abstractmethod
    def set_color(self, color: str):
        pass

    @abstractmethod
    def build(self) -> Car:
        pass
```

Le Builder contient toutes les méthodes permettant de créer l'objet étape par étape.

- **Le ConcreteBuilder**

Est nécessairement une classe fille du Builder abstrait qui va implémenter les méthodes de ce dernier.

Nous pouvons l'écrire dans le même fichier que celui du Builder :

```
class ConcreteCarBuilder(CarBuilder):  
    def __init__(self):  
        self.car = Car()  
  
    def set_make(self, make: str):  
        self.car.make = make  
        return self  
  
    def set_model(self, model: str):  
        self.car.model = model  
        return self  
  
    def set_year(self, year: int):  
        self.car.year = year  
        return self  
  
    def set_color(self, color: str):  
        self.car.color = color  
        return self  
  
    def build(self) -> Car:  
        return self.car
```

Une fois cela fait nous pouvons écrire une classe CarBuilderDirector qui ordonnera la construction du Product.

```
from Lib.classes.carbuilder import ConcreteCarBuilder
from Lib.classes.car import Car

class CarBuilderDirector:
    def __init__(self, builder: ConcreteCarBuilder):
        self._builder = builder

    def construct_metal_car(self, make: str, model: str, year: int):
        self._builder.set_make(make)
        self._builder.set_model(model)
        self._builder.set_year(year)
        return self._builder.build()

    def paint_car(self, car: Car, color: str):
        self._builder.set_make(car.make)
        self._builder.set_model(car.model)
        self._builder.set_color(color)
        return self._builder.build()

    def construct_finished_car(self, make: str, model: str, year: int,
color: str):
        self._builder.set_make(make)
        self._builder.set_model(model)
        self._builder.set_year(year)
        self._builder.set_color(color)
        return self._builder.build()
```

Le director n'est pas un composant obligatoire du motif de conception Builder. On pourrait utiliser directement le ConcreteBuilder pour ajouter progressivement des éléments à notre voiture.

```
Utilisation dans main.py
from Lib.utils import utils
from Lib.classes.carbuilder import ConcreteCarBuilder
from Lib.classes.carbuilderdirector import CarBuilderDirector

if __name__ == "__main__":
    print("\nUtilisation d'un Director")
    director = CarBuilderDirector(ConcreteCarBuilder())
    metalCar = director.contruct_metal_car("Toyota", "Corolla", 2024)
    utils.show(metalCar, 'metalCar')
```

```
Utilisation d'un Director
Objet observé: metalCar
Objet de classe: Car
```

```
Propriétés d'instance:
{'color': None, 'make': 'Toyota', 'model': 'Corolla', 'year': 2024}
```

```
Méthodes:
```

```
['describe']
```

```
print("-----")
print("On va maintenant peindre la voiture... -> director.paint_car")
print(metalCar, 'red'))
director.paint_car(metalCar, 'red')
utils.show(metalCar, 'metalCar')
metalCar.describe()
```

```
-----  
On va maintenant peindre la voiture... -> director.paint_car(metalCar, 'red')  
Objet observé: metalCar  
Objet de classe: Car  
  
Propriétés d'instance:  
{'color': 'red', 'make': 'Toyota', 'model': 2024, 'year': 2024}  
  
Méthodes:  
['describe']  
Car: 2024 Toyota 2024 in red
```

```
print("-----")  
print("Construisons une voiture complete : ")  
completedCar = director.construct_finished_car("Porshe", "Cayenne",  
2024, "black")  
completedCar.describe()
```

```
-----  
Construisons une voiture complete :  
Car: 2024 Porshe Cayenne in black
```

Nous pouvons également nous passer d'un Director.
En effet les méthodes du ConcreteDirector renvoient self ce qui permet de chainer les méthodes.

```
print("-----")  
print("Construisons une voiture en utilisant la méthode Fluent : ")  
builder = ConcreteCarBuilder()  
car = (builder  
    .set_make("Toyota")  
    .set_model("Corolla")  
    .set_year(2023)  
    .set_color("Blue")  
    .build())  
car.describe()
```

```
-----  
Construisons une voiture en utilisant la méthode Fluent :  
Car: 2023 Toyota Corolla in Blue
```

Builder Utilisant des Fonctions de Configuration

Une autre approche consiste à utiliser des fonctions ou des lambdas pour configurer le Builder, ce qui peut être particulièrement utile dans des contextes où la configuration est dynamique.

Tout d'abord, définissons le **Product** :

```
class Computer:  
    def __init__(self):  
        self.cpu = None  
        self.ram = None  
        self.storage = None  
        self.gpu = None  
  
    def describe(self):  
        print(f"Computer specs: CPU={self.cpu}, RAM={self.ram}GB, Storage={self.storage}GB, GPU={self.gpu}")
```

Puis le Builder Abstrait

```
from abc import ABC, abstractmethod
from Lib.classes import Computer

class ComputerBuilder(ABC):
    @abstractmethod
    def set_cpu(self, cpu: str):
        pass

    @abstractmethod
    def set_ram(self, ram: int):
        pass

    @abstractmethod
    def set_storage(self, storage: int):
        pass

    @abstractmethod
    def set_gpu(self, gpu: str):
        pass

    @abstractmethod
    def build(self) -> Computer:
        pass
```

Puis créons le ConcreteComputerBuilder dans le même fichier que le builder abstrait pour simplifier les imports.

```
from Lib.classes.computer import Computer
```

```
class ConcreteComputerBuilder(ComputerBuilder):
    def __init__(self):
        self.computer = Computer()

    def set_cpu(self, cpu: str):
        self.computer.cpu = cpu
        return self

    def set_ram(self, ram: int):
        self.computer.ram = ram
        return self

    def set_storage(self, storage: int):
        self.computer.storage = storage
        return self

    def set_gpu(self, gpu: str):
        self.computer.gpu = gpu
        return self

    def build(self) -> Computer:
        return self.computer
```

Et enfin créons le ComputerDirector.

```
ases / computerdirector.py / ComputerDirector / build_office_computer()
from Lib.classes.computerbuilder import ComputerBuilder

class ComputerDirector:
    def __init__(self, builder: ComputerBuilder):
        self.builder = builder

    def build_gaming_computer(self):
        self.builder.set_cpu("Intel i9") \
            .set_ram(32) \
            .set_storage(2000) \
            .set_gpu("NVIDIA RTX 3080")

    def build_office_computer(self):
        self.builder.set_cpu("Intel i5") \
            .set_ram(16) \
            .set_storage(500) \
            .set_gpu("Integrated Graphics")
```

Utilisons maintenant notre Builder pour construire un type de Computer.

```
print("-----")
print("Construisons un gaming computer : ")
builder = ConcreteComputerBuilder()
director = ComputerDirector(builder)

director.build_gaming_computer()
gaming_computer = builder.build()
gaming_computer.describe()
```

```
-----  
Construisons un gaming computer :  
Computer specs: CPU=Intel i9, RAM=32GB, Storage=2000GB, GPU=NVIDIA RTX 3080
```

Puis pour créer un autre type d'ordinateur on reset le Builder pour vider sa propriété interne.

```
print("-----")  
print("Construisons un office computer : ")  
builder = ConcreteComputerBuilder() # On reset le Builder  
director = ComputerDirector(builder)  
director.build_office_computer()  
office_computer = builder.build()  
office_computer.describe()
```

```
-----  
Construisons un office computer :  
Computer specs: CPU=Intel i5, RAM=16GB, Storage=500GB, GPU=Integrated Graphics
```

Réinitialisation du Builder : Pour créer un nouvel objet avec une configuration différente, une nouvelle instance de ConcreteComputerBuilder est utilisée.

Bonnes Pratiques pour Utiliser le Patron Builder

1. Évaluer la Nécessité :

- Utilisez le patron Builder principalement pour les objets complexes avec de nombreuses options ou configurations.

2. Séparer les Responsabilités :

- Assurez-vous que le Builder est uniquement responsable de la construction de l'objet et ne gère pas d'autres aspects de l'application.

3. Utiliser des Méthodes Fluent :

- Implémentez des méthodes qui renvoient self pour permettre un enchaînement fluide des appels de méthode, améliorant ainsi la lisibilité.

4. Encapsuler la Logique de Construction :

- Gardez la logique de construction au sein du Builder ou du Director, évitant de la disperser dans le code client.

NB :

Écrivez des tests unitaires pour les Builders afin de garantir qu'ils construisent les objets correctement selon les configurations spécifiées.

Comparaison avec d'autres Patrons de Crédation

Le patron Builder est l'un des plusieurs patrons de création en POO. Voici une comparaison avec d'autres patrons de création courants :

Patron	Objectif Principal	Utilisation Typique
Factory Method	Déléguer la création d'objets à des sous-classes.	Lorsque vous avez une hiérarchie de classes et que les sous-classes doivent décider quel objet créer.
Abstract Factory	Créer des familles d'objets liés sans spécifier leurs classes concrètes.	Lorsque vous devez créer des objets interconnectés et cohérents (ex : GUI pour différentes plateformes).
Singleton	Assurer qu'une classe n'a qu'une seule instance et fournir un point d'accès global.	Gestion des ressources partagées comme les connexions à une base de données ou les gestionnaires de log.
Builder	Construire des objets complexes étape par étape en séparant la construction de la représentation.	Création d'objets complexes avec de nombreuses configurations ou étapes, comme des configurations de produits ou des objets multimédias.
Prototype	Créer de nouveaux objets en copiant des objets existants (clonage).	Lorsque la création d'un objet est coûteuse et qu'il est plus efficace de le cloner.

Les Patrons STRUCTURELS

Adapter

Le patron de conception **Adapter**, également connu sous le nom de **Wrapper**, est un **patron structurel** qui permet à des classes avec des interfaces incompatibles de travailler ensemble. Il agit comme un pont entre deux interfaces en convertissant l'interface d'une classe en une interface que le client attend.

Analogies Métaphoriques :

- **Adaptateur électrique** : Permet à des appareils conçus pour une norme électrique spécifique de fonctionner avec une prise différente.
- **Traduction linguistique** : Un traducteur permet à deux personnes parlant des langues différentes de communiquer.

Caractéristiques Clés :

L'Adapter résout les problèmes où les interfaces des classes ne sont pas compatibles.

Scénarios Typiques d'Utilisation :

- Intégration de bibliothèques tierces avec des interfaces différentes de celles attendues par votre application.
- Adaptation d'anciennes classes ou de code legacy pour qu'ils fonctionnent avec de nouvelles interfaces.
- Création d'interfaces uniformes pour des systèmes hétérogènes.

Composants du patron Adapter

- La Cible (Target)
- Le Client (Client)
- L'Adaptateur (Adapter)
- L'Adapté (Adaptee)

1. Target

C'est l'interface que le client attend.

La cible définit les méthodes que le client utilise. Le client interagit avec le système via cette interface.

2. Le Client

C'est la classe ou la méthode qui utilise l'interface Target.

Elle interagit uniquement avec le Target, ignorant l'existence de l'Adapter et de l'Adapté.

3. L'Adaptateur

C'est la classe qui adapte l'interface de l'Adapté pour qu'elle soit compatible avec le Target.

Elle Implémente l'interface Target et détient une référence à l'Adapté. Elle traduit les appels du client en appels compatibles avec l'Adapté.

4. L'adapté

C'est la classe existante avec une interface incompatible avec le Target. Elle possède une interface spécifique que le client ne connaît pas ou ne peut pas utiliser directement.

Implémentation du Patron Adapter en Python

Voyons maintenant comment implémenter le patron Adapter en Python avec des exemples concrets.

Exemple 1 : Adapter d'Objet

Cet exemple montre comment utiliser l'Adapter en tant qu'objet qui adapte une classe existante pour qu'elle soit compatible avec une interface attendue.

Contexte :

Supposons que nous avons une classe Adaptee qui possède une méthode specific_request. Nous voulons que cette classe soit compatible avec une interface Target qui possède une méthode request.

Créons la classe **Adaptee**. Cette classe possède une méthode « `specific_request()` » incompatible avec le Target

```
class Adaptee:  
    def specific_request(self) -> str:  
        return ".eetpadA eht fo roivaherb  
        laicepS"
```

Créons le client dans `main.py`. Ce sera une méthode dans notre exemple. Le client utilise l'interface Target sans connaître l'Adapté ni l'Adapter.

```
main.py > ...  
from Lib.utils import utils  
from Lib.classes.target import Target  
  
def client_code(target: Target) -> None:  
    print(target.request())  
  
if __name__ == "__main__":
```

Créons la classe cible (Target) – classe abstraite

```
classes > {} target.py > ...  
from abc import ABC, abstractmethod  
  
class Target(ABC):  
    @abstractmethod  
    def request(self) -> str:  
        pass
```

Du fait du nommage de la méthode abstraite et celle que contient Adapté, on ne pourra pas fournir l'Adaptee au client car l'Adapté ne

possède pas le type Target à cause de la différence de nommage de sa méthode request.

Il nous faudra un transducteur entre les deux. Ce transducteur sera le rôle de la classe Adapter

Ecrivons l'Adapter. Il implémente le Target et détient une référence à l'Adaptee. Il traduit l'appel request() en appel à specific_request() et adapte le résultat.

```
from Lib.classes.target import Target
from Lib.classes.adaptee import Adaptee

class Adapter(Target):
    def __init__(self, adaptee: Adaptee):
        self._adaptee = adaptee

    def request(self) -> str:
        # Traduction de l'appel du client à l'Adaptee
        result = self._adaptee.specific_request()
        return f"Adapter: (TRANSLATED) {result[::-1]}"\n        # Retourne La\n        chaîne inversée
```

Utilisons ce patron dans notre main.py

```
from Lib.utils import utils
from Lib.classes.target import Target
from Lib.classes.adaptee import Adaptee
from Lib.classes.adapter import Adapter

def client_code(target: Target) -> None:
    print(target.request())

if __name__ == "__main__":
    adaptee = Adaptee()
    adapter = Adapter(adaptee)
    client_code(adapter)
```

Adapter: (TRANSLATED) Special behavior of the Adaptee.

Bonnes Pratiques pour Utiliser le Patron Adapter

1. Utiliser la Composition Plutôt que l'Héritage (si possible) :

- Privilégiez l'Adapter d'Objet pour réduire le couplage et augmenter la flexibilité.

2. Nommer Clairement les Classes :

- Utilisez des noms explicites comme XAdapter pour indiquer clairement la fonction de l'Adapter.

3. Minimiser les Changements dans l'Adapter :

- Gardez l'Adapter simple et évitez d'ajouter des fonctionnalités supplémentaires qui ne sont pas liées à l'adaptation.

4. Éviter les Adapters Cascadés :

- Évitez de chaîner plusieurs Adapters, ce qui peut compliquer la compréhension et la maintenance du code.

5. Tester les Adapters Indépendamment :

- Écrivez des tests unitaires pour les Adapters afin de garantir qu'ils traduisent correctement les appels entre le Target et l'Adaptee.

6. Considérer l'Utilisation de Patrons Similaires si Nécessaire :

- Parfois, d'autres patrons comme le Facade peuvent être plus appropriés selon le contexte.

Comparaison avec d'autres Patrons de Conception

Le patron Adapter est souvent confondu avec d'autres patrons de conception en raison de leurs similitudes. Voici une comparaison avec quelques-uns de ces patrons :

Patron	Objectif Principal	Similitude avec Adapter	Différence Clé
Facade	Fournir une interface simplifiée à un ensemble complexe de classes ou de sous-systèmes.	Simplifie l'interaction avec un ensemble de classes.	Focalisé sur la simplification de l'utilisation d'un système complet, pas sur l'adaptation des interfaces.
Proxy	Contrôler l'accès à un objet, souvent pour ajouter des fonctionnalités supplémentaires comme la sécurité ou la journalisation.	Peut intercepter les appels aux méthodes de l'objet cible.	Proxy vise à contrôler l'accès ou ajouter des fonctionnalités, tandis qu'Adapter vise à rendre deux interfaces compatibles.
Decorator	Ajouter dynamiquement des responsabilités supplémentaires à un objet sans modifier sa structure.	Peut modifier le comportement des méthodes de l'objet cible.	Decorator ajoute des fonctionnalités, Adapter transforme l'interface.
Bridge	Séparer l'abstraction d'une classe de son implémentation, permettant de les modifier indépendamment.	Facilite la flexibilité et l'extensibilité.	Bridge sépare abstraction et implémentation, Adapter transforme l'interface d'un objet.

Composite

Le patron de conception **Composite** est un **patron structurel** qui permet de composer des objets en structures arborescentes pour représenter des hiérarchies partie-tout. Ce patron permet aux clients de traiter de manière uniforme des objets individuels (feuilles) et des compositions d'objets (composites).

Quand vous considérez ce patron, vous pouvez imaginer la conception des éléments suivant comme illustrant le mécanisme sous jacent au patron composite :

Arbre Généalogique : Les individus (feuilles) et les familles (composites) peuvent être traités de la même manière pour naviguer dans l'arbre.

Organisation d'une Entreprise : Les employés (feuilles) et les départements (composites) peuvent être gérés de manière uniforme.

Caractéristiques Clés :

- **Uniformité** : Les objets simples et les compositions sont traités de la même manière.
- **Hiérarchie** : Permet de représenter des structures arborescentes complexes.
- **Transparence** : Les opérations peuvent être effectuées sur l'ensemble de la structure sans distinction.

Composants du patron Composite

- Le composant (component)
- La feuille (leaf)
- Le composite

1. Le composant (Component)

C'est une interface ou une classe abstraite qui définit les opérations communes pour les objets simples et les compositions d'objets.

Il est responsable de :

- Déclarer des opérations communes.
- Définir des méthodes pour ajouter, supprimer ou accéder aux composants enfants (optionnel).

Créons une classe abstraite nommée « Graphic »

```
classes > {} graphic.py > ...
from abc import ABC, abstractmethod

class Graphic(ABC):
    @abstractmethod
    def draw(self):
        pass

    @abstractmethod
    def add(self, graphic: 'Graphic'):
        pass

    @abstractmethod
    def remove(self, graphic: 'Graphic'):
        pass

    @abstractmethod
    def get_child(self, index: int) -> 'Graphic':
        pass
```

2. La Feuille (Leaf)

C'est un objet simple qui ne peut pas contenir d'autres objets. Il implémente les opérations définies par le Component.

Il est responsable de l'Implémentation des méthodes définies par le Composant.

Il ne doit pas contenir d'autres composants.

Créons une classe nommé « Dot »

```
from Lib.classes.graphic import Graphic

class Dot(Graphic):
    def draw(self):
        print("Drawing a dot.")

    def add(self, graphic: 'Graphic'):
        raise NotImplementedError("Cannot add to a leaf.")

    def remove(self, graphic: 'Graphic'):
        raise NotImplementedError("Cannot remove from a leaf.")

    def get_child(self, index: int) -> 'Graphic':
        raise NotImplementedError("Cannot get child from a leaf.")
```

Et une classe « Circle »

```
from Lib.classes.graphic import Graphic

class Circle(Graphic):
    def draw(self):
        print("Drawing a circle.")

    def add(self, graphic: 'Graphic'):
        raise NotImplementedError("Cannot add to a leaf.")

    def remove(self, graphic: 'Graphic'):
        raise NotImplementedError("Cannot remove from a leaf.")

    def get_child(self, index: int) -> 'Graphic':
        raise NotImplementedError("Cannot get child from a leaf.")
```

3. Le Composite

C'est un objet qui peut contenir d'autres objets (**Leaf ou Composite**). Il implémente les opérations définies par le Component et gère ses enfants.

Il est responsable de l'implémentation des méthodes pour ajouter, supprimer et accéder aux composants enfants, et de l'implémentation des opérations définies par le Component de manière récursive.

Créons une classe « CompoundGraphic »

```
from Lib.classes.graphic import Graphic

class CompoundGraphic(Graphic):
    def __init__(self):
        self.children = []

    def draw(self):
        print("Drawing a compound graphic:")
        for child in self.children:
            child.draw()

    def add(self, graphic: 'Graphic'):
        self.children.append(graphic)

    def remove(self, graphic: 'Graphic'):
        self.children.remove(graphic)

    def get_child(self, index: int) -> 'Graphic':
        return self.children[index]
```

Testons dans main.py :

```
from Lib.utils import utils
from Lib.classes.dot import Dot
from Lib.classes.circle import Circle
from Lib.classes.compoundgraphic import CompoundGraphic

if __name__ == "__main__":
    graph = CompoundGraphic()
    graph.add(Dot()) # On charge Dot dans le composite
    graph.add(Circle()) # on charge circle dans le composite
    utils.show(graph, 'graph')
    graph.draw()
    print("-----")
```

```
Objet observé: graph
Objet de classe: CompoundGraphic

Propriétés de classe:
{'_abc_impl': <_abc._abc_data object at 0x00000229D7585F40>}

Propriétés d'instance:
{'children': [<Lib.classes.dot.Dot object at 0x00000229D6FD6C90>,
              <Lib.classes.circle.Circle object at 0x00000229D71D5D10>]}

Méthodes:
['add', 'draw', 'get_child', 'remove']
Drawing a compound graphic:
Drawing a dot.
Drawing a circle.
-----
```

Exemple pratique

Supposons que nous voulons représenter un système de fichiers où les dossiers peuvent contenir des fichiers ou d'autres dossiers. Nous allons utiliser le patron Composite pour modéliser cette structure.

Composants du patron:

1. **Component** : FileSystemComponent
2. **Leaf** : File
3. **Composite** : Directory

Component

Interface abstraite définissant les méthodes `display`, `add`, `remove`, et `get_child`.

```
classes > {} filesystemcomponent.py > ...
from abc import ABC, abstractmethod

class FileSystemComponent(ABC):
    @abstractmethod
    def display(self, indent: str = ""):
        pass

    @abstractmethod
    def add(self, component: 'FileSystemComponent'):
        pass

    @abstractmethod
    def remove(self, component: 'FileSystemComponent'):
        pass

    @abstractmethod
    def get_child(self, index: int) -> 'FileSystemComponent':
        pass
```

Leaf

Classe Leaf représentant un fichier. Elle implémente display pour afficher son nom et lève des exceptions pour les méthodes add, remove, et get_child car un fichier ne peut pas contenir d'autres composants.

```
classes > {} file.py > ...
from Lib.classes.filesystemcomponent import FileSystemComponent

class File(FileSystemComponent):
    def __init__(self, name: str):
        self.name = name

    def display(self, indent: str = ""):
        print(f"{indent}File: {self.name}")

    def add(self, component: 'FileSystemComponent'):
        raise NotImplementedError("Cannot add to a file.")

    def remove(self, component: 'FileSystemComponent'):
        raise NotImplementedError("Cannot remove from a file.")

    def get_child(self, index: int) -> 'FileSystemComponent':
        raise NotImplementedError("Cannot get child from a file.")
```

Composite

Classe Composite représentant un dossier. Elle contient une liste de FileSystemComponent et implémente les méthodes add, remove, get_child, et display de manière récursive pour afficher toute la hiérarchie.

```
from Lib.classes.filesystemcomponent import FileSystemComponent
from typing import List

class Directory(FileSystemComponent):
    def __init__(self, name: str):
        self.name = name
        self.children: List[FileSystemComponent] = []

    def display(self, indent: str = ""):
        print(f"{indent}Directory: {self.name}")
        for child in self.children:
            child.display(indent + "  ")

    def add(self, component: 'FileSystemComponent'):
        self.children.append(component)

    def remove(self, component: 'FileSystemComponent'):
        self.children.remove(component)

    def get_child(self, index: int) -> 'FileSystemComponent':
        return self.children[index]
```

Dans main.py :

```
from Lib.classes.file import File
from Lib.classes.directory import Directory
```

```
print("-----")
done = "Fait!"

print("Utilisation du File system Component")
print("Création de 3 fichiers...")
file1 = File("file1.txt")
file2 = File("file2.txt")
file3 = File("file3.txt")
print(done)

print("Création de 3 dossiers...")
dir1 = Directory("dir1")
dir2 = Directory("dir2")
dir3 = Directory("dir3")
print(done)

print("Construction de la hiérarchie...")
dir1.add(file1)
dir1.add(file2)

dir2.add(file3)
dir2.add(dir3)
print(done)

print("Construction du dossier racine...")
root = Directory("root")
root.add(dir1)
root.add(dir2)
print(done)

print("Affichage de la structure du file system...")
root.display()
```

```

Utilisation du File system Component
Création de 3 fichiers...
Fait!
Création de 3 dossiers...
Fait!
Construction de la hiérarchie...
Fait!
Construction du dossier racine...
Fait!
Affichage de la structure du file system...
Directory: root
    Directory: dir1
        File: file1.txt
        File: file2.txt
    Directory: dir2
        File: file3.txt
    Directory: dir3

```

Comparaison avec d'autres Patrons de Conception

Patron	Objectif Principal	Similitude avec Composite	Différence Clé
Decorator	Ajouter dynamiquement des responsabilités supplémentaires à un objet sans modifier sa structure.	Permet de traiter des objets de manière uniforme.	Decorator modifie le comportement des objets, tandis que Composite crée des structures arborescentes.
Facade	Fournir une interface simplifiée à un ensemble complexe de classes ou de sous-systèmes.	Facilite l'interaction avec une structure complexe.	Facade simplifie l'utilisation d'un système, Composite organise des objets en hiérarchie.
Adapter	Permettre à des classes avec des interfaces incompatibles de travailler ensemble.	Facilite l'interopérabilité entre composants.	Adapter transforme l'interface d'un objet, Composite crée une structure hiérarchique.
Strategy	Définir une famille d'algorithme, encapsuler chacun d'eux, et les rendre interchangeables.	Aucun lien direct.	Strategy encapsule des comportements interchangeables, tandis que Composite organise des objets.

Le Décorateur

Le patron de conception **Decorator** est un **patron structurel** qui permet d'ajouter de manière dynamique des fonctionnalités supplémentaires à des objets individuels sans affecter les autres objets de la même classe. Il agit comme un **enveloppant** qui contient l'objet à décorer et ajoute ou modifie son comportement.

Vous pouvez ajouter différents vêtements à une personne (par exemple, un manteau, un chapeau) sans modifier la personne elle-même.

De même, vous pouvez ajouter un ruban ou un papier cadeau à un cadeau sans changer le cadeau lui-même.

Caractéristiques Clés

- **Flexibilité** : Permet d'ajouter ou de retirer des responsabilités à la volée.
- **Encapsulation** : Les responsabilités supplémentaires sont encapsulées dans les décorateurs, séparant ainsi les fonctionnalités additionnelles de la logique de base.

Le patron Decorator suit une structure définie qui inclut plusieurs composants interconnectés. Comprendre ces composants est essentiel pour implémenter efficacement le patron.

Composants du patron Decorator

- Le component
- Le ConcreteComponent
- Le Decorator
- Le concreteDecorator

1. Component

C'est une interface ou une classe abstraite qui définit les méthodes communes pour les objets à décorer.

Il a pour responsabilité de déclarer les méthodes que les décorateurs et les composants concrets doivent implémenter.

Créer un classe « Beverage » (abstraite)

```
classes > {} beverage.py > ...
from abc import ABC, abstractmethod

class Beverage(ABC):
    @abstractmethod
    def get_description(self) -> str:
        pass

    @abstractmethod
    def cost(self) -> float:
        pass
```

2. ConcreteComponent

C'est une classe qui implémente l'interface Component et représente les objets que l'on souhaite décorer.

Elle est responsable d'Implémenter les méthodes définies dans Component.

```
asses > {} expresso.py > ...
from Lib.classes.beverage import Beverage

class Espresso(Beverage):
    def get_description(self) -> str:
        return "Espresso"

    def cost(self) -> float:
        return 1.99
```

3. Decorator

C'est une classe abstraite qui implémente l'interface Component et contient une référence à un objet Component. Elle délègue les appels de méthodes au composant qu'elle décore.

Elle a pour responsabilité de :

- Contenir une référence à un objet Component.
- Définir une interface pour les décorateurs concrets.

```
classes > {} condimentdecorator.py > ...
```

```
from abc import ABC, abstractmethod
from Lib.classes.beverage import Beverage

class CondimentDecorator(Beverage, ABC):
    def __init__(self, beverage: Beverage):
        self.beverage = beverage

    @abstractmethod
    def get_description(self) -> str:
        pass
```

4. ConcreteDecorator

C'est une classe qui hérite de Decorator et ajoute des fonctionnalités supplémentaires au composant.

Elle est responsable d'ajouter des comportements ou des propriétés supplémentaires ainsi que d'implémenter les méthodes de Component, souvent en appelant les méthodes du composant décoré et en ajoutant des fonctionnalités.

```
classes > {} mocha.py > ...
from Lib.classes.condimentdecorator import CondimentDecorator

class Mocha(CondimentDecorator):
    def get_description(self) -> str:
        return f"{self.beverage.get_description()}, Mocha"

    def cost(self) -> float:
        return self.beverage.cost() + 0.20
```

```
classes > {} milk.py > ...
from Lib.classes.condimentdecorator import CondimentDecorator

class Milk(CondimentDecorator):
    def get_description(self) -> str:
        return f"{self.beverage.get_description()}, Milk"

    def cost(self) -> float:
        return self.beverage.cost() + 0.10
```

```
classes > {} sugar.py > ...
from Lib.classes.condimentdecorator import CondimentDecorator

class Sugar(CondimentDecorator):
    def get_description(self) -> str:
        return f"{self.beverage.get_description()}, Sugar"

    def cost(self) -> float:
        return self.beverage.cost() + 0.05
```

Utilisons dans main.py

```
from Lib.utils import utils
from Lib.classes.expresso import Espresso
from Lib.classes.milk import Milk
from Lib.classes.sugar import Sugar

if __name__ == "__main__":
    beverage = Espresso()
    beverage = Milk(beverage)
    beverage = Sugar(beverage)

    print(f"{beverage.get_description()} - ${beverage.cost():.2f}")
```

```
[nodemon] starting `python main.py`
Espresso, Milk, Sugar - $2.14
[nodemon] clean exit - waiting for change
```

Création de décorateur custom dans Python

En Python, le terme "decorator" est également utilisé pour désigner les fonctions décoratrices, qui sont une fonctionnalité du langage permettant d'ajouter des fonctionnalités à des fonctions ou des méthodes de manière concise.

Bien que cela diffère légèrement du patron de conception Decorator, il partage des concepts similaires d'ajout de responsabilités.

Allons dans Lib/utils et créer un fichier decorators.py :

```
> utils > {} decorators.py > ...
def bold_decorator(func):
    def wrapper():
        return f"<b>{func()}</b>"
    return wrapper

def italic_decorator(func):
    def wrapper():
        return f"<i>{func()}</i>"
    return wrapper
```

Puis utilisons ces décorateurs dans main.py :

```
from Lib.utils.decorators import bold_decorator, italic_decorator
```

```
print("-----")
```

```
@bold_decorator  
@italic_decorator  
def saluer():  
    return "Hello World"
```

```
print(saluer())
```

```
-----
```

```
<b><i>Hello World</i></b>
```

Les décorateurs vont être exécutés de bas en haut.

Patrons de conception Comportementaux

Observer

Le patron de conception **Observer** est un **patron comportemental** qui permet de définir une relation de dépendance entre des objets de telle sorte que lorsqu'un objet (le **Sujet**) change d'état, tous les objets qui en dépendent (les **Observateurs**) sont automatiquement notifiés et mis à jour.

Par exemple, une station météo (Sujet) envoie des mises à jour sur les conditions météorologiques à plusieurs applications ou dispositifs (Observateurs) qui affichent ces informations. Ou encore, dans le cadre des réseaux sociaux, lorsqu'un utilisateur publie une nouvelle photo (Sujet), tous ses abonnés (Observateurs) reçoivent une notification.

Caractéristiques Clés

- **Notification Automatique** : Les Observateurs sont automatiquement informés des changements d'état du Sujet.

Scénarios Typiques d'Utilisation

- **Interfaces Utilisateur** : Mise à jour automatique des éléments de l'interface lorsque les données sous-jacentes changent.
- **Systèmes de Notification** : Envoi de notifications à plusieurs systèmes ou utilisateurs lorsqu'un événement se produit.
- **Flux de Données en Temps Réel** : Synchronisation des données entre plusieurs composants ou services.

Composants du patron Observer

- Le Sujet (Subject)
- L'Observateur (Observer)

Subject

Le Sujet maintient une liste d'Observateurs et les notifie automatiquement de tout changement d'état.

Il est responsable d'ajouter et supprimer des Observateurs et de notifier tous les Observateurs en cas de changement d'état.

Créer une classe Subject (abstraite)

```
classes > {} subject.py > ...
from abc import ABC, abstractmethod

class Subject(ABC):
    @abstractmethod
    def attach(self, observer: 'Observer'):
        pass

    @abstractmethod
    def detach(self, observer: 'Observer'):
        pass

    @abstractmethod
    def notify(self):
        pass
```

Observer

Les Observateurs définissent une interface pour recevoir des notifications du Sujet. Ils reçoivent et traitent les notifications du Sujet.

Créer une classe Observer :

```
lasses > {} observer.py > {} Observer > {} update
from abc import ABC, abstractmethod
from Lib.classes.subject import Subject

class Observer(ABC):
    @abstractmethod
    def update(self, subject: Subject):
        pass
```

Créons maintenant un Subject concret

```
classes > {} wxstation.py > ...
from Lib.classes.subject import Subject
from Lib.classes.observer import Observer
from typing import List
|
class ConcreteWeatherStation(Subject):
    def __init__(self):
        self._observers: List[Observer] = []
        self._temperature: float = 0.0
        self._humidity: float = 0.0

    def attach(self, observer: 'Observer'):
        if observer not in self._observers:
            self._observers.append(observer)
            print(f"Observer {observer.__class__.__name__} attaché.")

    def detach(self, observer: 'Observer'):
        try:
            self._observers.remove(observer)
            print(f"Observer {observer.__class__.__name__} détaché.")
        except ValueError:
            pass

    def notify(self):
        print("Notification des Observateurs...")
        for observer in self._observers:
            observer.update(self)

    def set_measurements(self, temperature: float, humidity: float):
        self._temperature = temperature
        self._humidity = humidity
        print(f"\nMesures mises à jour: Température = {self._temperature}°C, Humidité = {self._humidity}%")
        self.notify()

    @property
    def temperature(self) -> float:
        return self._temperature

    @property
    def humidity(self) -> float:
        return self._humidity
```

Créons maintenant deux classes de type Observer

```
from Lib.classes.observer import Observer
from Lib.classes.subject import Subject

class MobileApp(Observer):
    def update(self, subject: Subject):
        print(f"MobileApp: Affichage des nouvelles mesures -> Température: {subject.temperature}°C, Humidité: {subject.humidity}%")
```

```
from Lib.classes.observer import Observer
from Lib.classes.subject import Subject
from Lib.classes import Subject

class DisplayBoard(Observer):
    def update(self, subject: Subject):
        print(f"DisplayBoard: Mise à jour de l'affichage -> Température: {subject.temperature}°C, Humidité: {subject.humidity}%")
```

Puis utilisons dans main.py

```
from Lib.utils import utils
from Lib.classes.wxstation import ConcreteWeatherStation
from Lib.classes.mobileapp import MobileApp
from Lib.classes.displayboard import DisplayBoard

if __name__ == "__main__":
    #instanciation d'un Subject
    subject = ConcreteWeatherStation()

    # Création des Observateurs
    mobile_app = MobileApp()
    display_board = DisplayBoard()

    # Attachement des Observateurs
    subject.attach(mobile_app)
    subject.attach(display_board)

    # Mise à jour des mesures météorologiques
    subject.set_measurements(25.3, 65.0)
    subject.set_measurements(26.7, 70.0)

    # Déattachement d'un Observateur
    subject.detach(mobile_app)

    # Nouvelle mise à jour des mesures
    subject.set_measurements(24.8, 60.0)
```

```
Observer MobileApp attaché.  
Observer DisplayBoard attaché.
```

```
Mesures mises à jour: Température = 25.3°C, Humidité = 65.0%  
Notification des Observateurs...  
MobileApp: Affichage des nouvelles mesures -> Température: 25.3°C, Humidité: 65.0%  
DisplayBoard: Mise à jour de l'affichage -> Température: 25.3°C, Humidité: 65.0%
```

```
Mesures mises à jour: Température = 26.7°C, Humidité = 70.0%  
Notification des Observateurs...  
MobileApp: Affichage des nouvelles mesures -> Température: 26.7°C, Humidité: 70.0%  
DisplayBoard: Mise à jour de l'affichage -> Température: 26.7°C, Humidité: 70.0%  
Observer MobileApp détaché.
```

```
Mesures mises à jour: Température = 24.8°C, Humidité = 60.0%  
Notification des Observateurs...  
DisplayBoard: Mise à jour de l'affichage -> Température: 24.8°C, Humidité: 60.0%
```

Comparaison avec d'autres Patrons de Conception

Patron	Objectif Principal	Similitude avec Observer	Différence Clé
Mediator	Faciliter la communication entre objets sans qu'ils aient besoin de se référer directement les uns aux autres.	Facilite la communication entre objets.	Mediator centralise la communication, tandis qu'Observer gère les dépendances de notification.
Strategy	Définir une famille d'algorithmes, les encapsuler et les rendre interchangeables.	Aucun lien direct.	Strategy encapsule des comportements interchangeables, Observer gère les dépendances de notification.
Command	Encapsuler une requête en tant qu'objet, permettant de paramétrier des clients avec des requêtes différentes.	Aucun lien direct.	Command encapsule des actions, Observer gère les dépendances de notification.
Pub/Sub	Permettre aux émetteurs de publier des messages sans connaître les abonnés.	Similaire à Observer avec une couche d'abstraction supplémentaire.	Pub/Sub utilise un intermédiaire (broker) pour gérer les messages, Observer établit une relation directe.

Strategy

Le patron de conception **Strategy** est un **patron comportemental** qui permet de définir une famille d'algorithmes, de les encapsuler chacun dans une classe distincte et de les rendre interchangeables.

Le patron **Strategy** permet de varier l'algorithme indépendamment des clients qui l'utilisent.

Par exemple, dans un jeu d'échec, différentes stratégies de jeu (attaque, défense, contrôle du centre) peuvent être sélectionnées en fonction de la situation sans modifier les pièces elles-mêmes.

Également, dans le cadre de la navigation GPS, différents itinéraires (le plus court, le plus rapide, éviter les péages) peuvent être choisis sans changer l'appareil GPS.

Caractéristiques Clés

- **Encapsulation des Algorithmes** : Chaque stratégie est encapsulée dans une classe distincte.
- **Interchangeabilité** : Les stratégies peuvent être échangées facilement sans modifier le code client.
- **Ouverture aux Extensions** : Facile d'ajouter de nouvelles stratégies sans changer le code existant.

Composants du patron Strategy

- Strategy
- ConcreteStrategy
- Context

Strategy

C'est une interface commune pour toutes les stratégies concrètes. Elle déclare une méthode que toutes les stratégies doivent implémenter.

Elle est responsable de définir une méthode commune que les stratégies concrètes doivent implémenter.

Créer une classe abstraite nommée « CompressionStrategy »

```
classes > {} compressionstrategy.py > ...
from abc import ABC, abstractmethod

class CompressionStrategy(ABC):
    @abstractmethod
    def compress(self, data: str) -> str:
        pass
```

ConcreteStrategy

Ce sont des classes concrètes qui implémentent l'interface Strategy avec des comportements spécifiques.

```
classes > {} rarstrategy.py > ...
from Lib.classes.compressionstrategy import CompressionStrategy

class RarCompressionStrategy(CompressionStrategy):
    def compress(self, data: str) -> str:
        return f"Compressed using RAR: {data[:10]}..."
```

```
classes > {} zipstrategy.py > ...
from Lib.classes.compressionstrategy import CompressionStrategy

class ZipCompressionStrategy(CompressionStrategy):
    def compress(self, data: str) -> str:
        return f"Compressed using ZIP: {data[:10]}..."
```

Context

C'est la classe qui utilise une instance de Strategy. Elle n'a pas besoin de connaître les détails de l'algorithme utilisé.

Elle est responsable de maintenir une référence à une instance de Strategy et d'appeler la méthode de l'algorithme via l'interface Strategy.

```
classes > compressioncontext.py > ...
from Lib.classes.compressionstrategy import CompressionStrategy

class CompressionContext:
    def __init__(self, strategy: CompressionStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: CompressionStrategy):
        self._strategy = strategy

    def create_archive(self, data: str) -> str:
        return self._strategy.compress(data)
```

Utilisons notre patron dans main.py

```
main.py > ...
from Lib.utils import utils
from Lib.classes.zipstrategy import ZipCompressionStrategy
from Lib.classes.rarstrategy import RarCompressionStrategy
from Lib.classes.compressioncontext import CompressionContext

if __name__ == "__main__":
    data = "This is some example data that needs to be compressed."
    # Initialisation avec la stratégie ZIP
    context = CompressionContext(ZipCompressionStrategy())
    print(context.create_archive(data))

    # Changement de stratégie à RAR
    context.set_strategy(RarCompressionStrategy())
    print(context.create_archive(data))
```

Compressed using ZIP: This is so...
Compressed using RAR: This is so...

FIN DE LA 2^{nde} PARTIE

